CORA Cork Open Research Archive
Cartlann Taighde Oscailte Chorcaí

UCC Library and UCC researchers have made this item openly available. Please let us know how this has helped you. Thanks!

Title	Real-time algorithm configuration
Author(s)	Fitzgerald, Tadhg
Publication date	2021-04
Original citation	Fitzgerald, T. 2021. Real-time algorithm configuration. PhD Thesis, University College Cork.
Type of publication	Doctoral thesis
Rights	**© 2021, Tadhg Fitzgerald.** https://creativecommons.org/licenses/by-nc-nd/4.0/
Item downloaded from	http://hdl.handle.net/10468/11303

Downloaded on 2021-11-27T16:37:32Z

UCC
University College Cork, Ireland
Coláiste na hOllscoile Corcaigh

Real-time Algorithm Configuration

Tadhg Fitzgerald

NATIONAL UNIVERSITY OF IRELAND, CORK

COLLEGE OF SCIENCE, ENGINEERING, AND FOOD SCIENCE

SCHOOL OF COMPUTER SCIENCE AND INFORMATION TECHNOLOGY

INSIGHT CENTRE FOR DATA ANALYTICS

Thesis submitted for the degree of
Doctor of Philosophy

April 2021

Supervisors: Prof. Barry O'Sullivan
Prof. Ken Brown
Head of Department/School: Prof. Cormac J. Sreenan

Research supported by Insight Centre for Data Analytics and Science Foundation Ireland under Grant No. 12/RC/2289 which is co-funded under the European Regional Development Fund.

Contents

List of Figures

List of Tables

Abstract

This dissertation presents a number of contributions to the field of algorithm configuration. In particular, we present an extension to the algorithm configuration problem, *real-time algorithm configuration*, where configuration occurs online on a stream of instances, without the need for prior training, and problem solutions are returned in the shortest time possible. We propose a framework for solving the *real-time algorithm configuration* problem, *ReACT*. With *ReACT* we demonstrate that by using the parallel computing architectures, commonplace in many systems today, and a robust aggregate ranking system, configuration can occur without any impact on performance from the perspective of the user. This is achieved by means of a racing procedure. We show two concrete instantiations of the framework, and show them to be on a par with or even exceed the state-of-the-art in offline algorithm configuration using empirical evaluations on a range of combinatorial problems from the literature.

We discuss, assess, and provide justification for each of the components used in our framework instantiations. Specifically, we show that the TrueSkill ranking system commonly used to rank players' skill in multiplayer games can be used to accurately estimate the quality of an algorithm's configuration using only censored results from races between algorithm configurations. We confirm that the order that problem instances arrive in influences the configuration performance and that the optimal selection of configurations to participate in races is dependent on the distribution of the incoming instance stream. We outline how to maintain a pool of quality configurations by removing underperforming configurations, and techniques to generate replacement configurations with minimal computational overhead. Finally, we show that the configuration space can be reduced using feature selection techniques from the machine learning literature, and that doing so can provide a boost in configuration performance.

Declaration

This dissertation is submitted to University College Cork, in accordance with the requirements for the degree of Doctor of Philosophy in the Faculty of Science. The research and thesis presented in this dissertation are entirely my own work and have not been submitted to any other university or higher education institution, or for any other academic award in this university. Where use has been made of other people's work, it has been fully acknowledged and referenced. Parts of this work have appeared in the following publications which have been subject to peer review:

- Tadhg Fitzgerald, Barry O'Sullivan, Yuri Malitsky, and Kevin Tierney. Online search algorithm configuration. In Carla E. Brodley and Peter Stone, editors, *Proceedings of the Twenty-Eighth AAAI Conference on Artificial Intelligence, July 27 -31, 2014, Québec City, Québec, Canada*, pages 3104–3105. AAAI Press, 2014.

- Tadhg Fitzgerald, Yuri Malitsky, Barry O'Sullivan, and Kevin Tierney. ReACT: Real-Time Algorithm Configuration through Tournaments. In Stefan Edelkamp and Roman Barták, editors, *Proceedings of the Seventh Annual Symposium on Combinatorial Search, SOCS 2014, Prague, Czech Republic, 15-17 August 2014*. AAAI Press, 2014.

- Tadhg Fitzgerald, Yuri Malitsky, and Barry O'Sullivan. ReACTR: Realtime Algorithm Configuration through Tournament Rankings. In Qiang Yang and Michael J. Wooldridge, editors, *Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence, IJCAI 2015, Buenos Aires, Argentina, July 25-31, 2015*, pages 304–310. AAAI Press, 2015.

- Tadhg Fitzgerald and Barry O'Sullivan. Analysing the effect of candidate selection and instance ordering in a realtime algorithm configuration system. In Ahmed Seffah, Birgit Penzenstadler, Carina Alves, and Xin Peng, editors, *Proceedings of the Symposium on Applied Computing, SAC 2017, Marrakech, Morocco, April 3-7, 2017*, pages 1003–1008. ACM, 2017.

- Tadhg Fitzgerald and Barry O'Sullivan. Candidate selection and instance ordering for realtime algorithm configuration. *Fundam. Informaticae*, 166(2):141–166, 2019.

The contents of this dissertation extensively elaborate upon previously published work and mistakes (if any) are corrected.

Tadhg Fitzgerald

April 2021

Acknowledgements

First and foremost, I would like to express my sincere gratitude to my PhD supervisor Prof. Barry O'Sullivan his unwavering support and encouragement throughout my doctoral studies. Barry's expert guidance, patience and constructive criticism has pushed me to become a better researcher and for this I am extremely grateful. I would also like to thank second supervisor Prof. Ken Brown for his invaluable advice and discussions throughout my PhD journey.

I must also thank my advisor Dr. Yuri Malitsky for guiding me along the right path at the beginning of my studies and for suggesting algorithm configuration as a thesis topic. I also owe a debt of gratitude to Helmut Simonis and Dr. Chrys Ngwa who I have had the pleasure of working on a number of industry projects with. Helmut has thought me so much about how to approach data science for an industrial project, while Chrys, in addition to cheerleading my thesis writing efforts, has given me so much helpful advice about the soft skills needed for business.

I am deeply indebted to my thesis examiners, Dr. Nikola Nikolov and Dr. Alejandro Arbelaez, whose thorough proofreading and thoughtful comments have helped improve the clarity of this thesis.

My time as a researcher in the Insight Centre has been enhanced by a number of great people that I have been lucky enough to work alongside. This acknowledgement section is far too short to name everyone who deserves a mention but I would like to thank all of my lab mates for the conversations, laughs, and support that they have provided during my studies. In particular I would like to thank Barry Hurley, Diarmuid Grimes, Ignacio Castiñeiras, Milan De Cauwer, Cathal Hoare, Yves Sohege, Begüm Genç, Diego Carraro, Andrea Visentin, Federico Toffano, Daniel Desmond, Sorina Chisca, Anne-Marie George, Mohamed Siala, Lars Kotthoff, Gilles Simonin, Bastien Pietropaoli, and all the rest of the bridge tea break crew. I would also like to express my sincere gratitude to the administrative team at Insight, Caitríona Walsh, Eleanor O'Riordon, Peter McHale and Linda O'Sullivan, for making everything seamless.

I would also like to thank the School of Computer Science in UCC who have supported

and helped me in numerous ways throughout both my undergraduate and doctoral studies.

Finally, I would like to thank my family. My parents, Ann and Tim, have provided incredible love, encouragement and support throughout my life and especially during the challenging writing phase of this PhD. My sister, Niamh, also deserves my deepest gratitude for her invaluable advice and unwavering support.

This dissertation would not have been possible without the financial support of Science Foundation Ireland Grants No. 12/RC/2289-P1 and No. 12/RC/2289-P2 which are co-funded under the European Regional Development Fund.

Chapter 1

Introduction

1.1 Motivation

Complexity surrounds us in the modern world. Many difficult modern challenges can be modelled as discrete optimisation problems. For example the routes taken by taxis and delivery trucks are often planned by routing software in order to minimise fuel usage and delay to the customer [CAG+14]. Internet advertising, spectrum sales and parcels of land are sold by means of combinatorial auctions to maximise seller profit and utility to the buyer [HIK+18, BG17, LBPS00b]. The integrated circuits used in all types of electronics use specialised software to verify that they are correct and will not malfunction [PBG05].

These problems and many more leverage the power of modern combinatorial solvers to provide optimal or near optimal solutions. Problems such as these are often extremely computationally challenging to solve as the solving difficulty usually increases exponentially with the number of options available. Due to the exponential explosion in size of the search space associated with these problems it is often impossible to completely search through all search states in order to find provably optimal solutions. For this reason the combinatorial solvers used to tackle these problems rely on various heuristics and problem relaxations to find good solutions. Due to the level of sophistication in these solvers the creators often expose a vast array of options as parameters [GO20, IBM14]. These parameters control everything from the search heuristic to the number of cuts to make.

It is now increasingly recognised that there is no single solver or parameter setting that works best on every type of instance [XHHL12]. Instead, different problems are best solved by different strategies, meaning solvers should open parameters to the

end-user in order to achieve maximum performance [Hoo12a]. By exposing an array of solver settings we are given a level of fine grained control that allows solvers be tailored to individual problems. This level of customisation can drastically improve the performance of high performance algorithms. However, this flexibility also exposes the end user to an overwhelming number of choices when configuring the solver. Even a domain expert with years of experience would struggle to select the best, or even a good, configuration given the myriad of different parameter settings and the potential interactions between them. Manual tuning and testing of anything but a small fraction of the entire configuration space is impossible due to its vastness.

For this reason there has been an emphasis on the field of automatic algorithm configuration in recent years [HHLBS09, HHL11, AST09b, AMS+15, BYBS10]. Automatic algorithm configuration, also known as parameter tuning, as the name implies aims to configure a parameterised algorithm such that its performance (e.g. runtime, objective value etc.) over a set of instances is improved. Algorithm configuration is usually performed in a black box fashion where the configurator is only supplied with a description of the algorithm parameters and their allowed values, a set or stream of instances to solve and a performance metric for the algorithm. The algorithm configurator then attempts to automatically find a set of legal values for the parameters (known as a configuration) such that performance on the supplied instances is increased.

The majority of current algorithm configuration systems treat tuning as a static problem [HHLBS09, HHL11, AST09b, AMS+15]. We refer to these types of algorithm configuration systems as offline configurators. An offline configurator is supplied with a set of representative training instances, an algorithm to tune and performance metric (e.g. runtime). The configurator is then allocated a fixed configuration budget of time which is used to train on the instances with the aim of producing a superior configuration. Within this configuration budget the offline configurator generates and evaluates configurations on the set of training instances. As training usually occurs offline and separate from the solving step it is possible to use techniques not available in situations where finding solution as quickly as possible is the main goal. For example, it is possible to revisit and solve the same instance multiple times in order to reduce variance and evaluate different configurations [HHLBS09, HHL11, AST09b, AMS+15, BYBS10]. A fixed configuration budget also enables more expensive techniques such as expensive instance feature computation and computationally intensive learning techniques to be used.

A number of offline algorithm configuration systems that employ many different techniques in order to seek out quality configurations have been introduced. Para-

mILS [HHLBS09], for example, employs an iterated local search to explore the parameter space, focussing on areas where it found improvements in the past. Alternatively, SMAC [HHL11] tries to build an internal random forest model that predicts the performance of a configuration, trying the ones most probable to improve upon the current behavior. Finally, GGA [AST09b] utilises a genetic approach, running a number of configurations in parallel and allowing the best ones to pass on their parameter settings to the subsequent generation. While there is no consensus on which of these approaches is best, all of them have been repeatedly validated in practice, sometimes leading to orders of magnitude improvements over what was found by human experts [HHL11].

While this methodology has been shown to work time and again sometimes resulting in orders of magnitude improvement over the default configuration [HHLBS09], it is not without its flaws. Firstly a representative set of training instances must be readily available before configuration can begin. Due to this requirement a suitable number of instances must be collected during the initial algorithm run before offline configuration can proceed. During this collection period no parameter tuning occurs and the default algorithm configuration must be used when solving instances. Alternatively, previously collected instances from a similar domain or problem type may be used as training instances. In this case the quality of the configured algorithm depends strongly on how closely the training instances match the instances encountered in the future. Differences in problem structure can result in drastically different parameter settings being preferred [Hoo12b, KHNT19].

In cases where a set of representative problem instances are available at training time there is no guarantee that these instances will remain representative as future instances grow and change over time. This is a common situation; consider an advertising company which runs combinatorial auctions to sell advertising space. As the company grows the customer demand and available advertising spaces change. Similarly the number of bids increases during times of high demand, for example in the run up to Christmas. Due to the train-once nature of offline configurators it is not possible to adapt to these changes without costly retraining. Even when retraining is possible it is not immediately obvious when retraining should occur. Periodically retraining allows the configurator to adapt to changes in the problem instances but comes with a time and computational penalty.

Finding the balance between configuration budget and improvement in solving time is a challenge. In the case of some problem instances that are not soluble in a reasonable amount of time, substantial algorithm configuration is needed to solve those instances efficiently. In other cases where the configuration budget is large and the improvement

is relatively modest, any solving time reduction through configuration can be eclipsed by the time taken to find a quality configuration. To the best of our knowledge the topic of balancing configuration budget with expected improvement has not been studied.

This thesis introduces a novel take on the classic algorithm configuration problem, real-time algorithm configuration, with the goal of resolving the aforementioned issues. The core objective of real-time algorithm configuration is to reduce time needed to return a solution while still improving the algorithm configuration over time. Real-time algorithm configuration works with a stream of instances, constantly improving the algorithm's configuration as the stream is being processed. Should the instances in the stream evolve over time the real-time configurator is able to adapt as it is constantly tuning. This is in stark contrast to offline configuration which adopts a train-once methodology and requires retraining when instances change. A streaming approach to configuration also alleviates the necessity of having a pre-existing training set to hand. Our proposed solution uses a lightweight techniques and a solve-once approach to ensure that an improving set of configurations are discovered while incurring the minimum amount of additional runtime.

There are numerous discrete optimisation challenges where instances naturally arrive in streams. Taxi and ride-sharing services must route drivers to pickups in a timely manner while a stream of requests arrivers from users [GRW08, LCH+14]. Combinatorial auctions are used to auction online advertising space, service procurement, and material supply [DVV03, HIK+18]. Here combinatorial auctions are repeatedly solved as bids change. Factories use scheduling techniques to schedule workers and machines, as worker availability and demand shifts schedules must be recalculated [CD09]. All of these practical applications would benefit heavily from the increases in efficiency which algorithm configuration provides. Despite this there has been little to no research into algorithm configurators which work on an incoming stream of instances in real-time [KHNT19].

The realisation of our work on real-time algorithm configuration is the Real-time Algorithm Configuration through Tournaments (ReACT) framework. The ReACT framework consists of four main components: racing, ranking/selection, removal, and generation. ReACT uses the multiple cores which are commonly available in modern architectures to race competing configurations in parallel. As soon a solution for the instance is found all other runs are terminated. This aggressive capping mechanism ensures that only the minimum evaluation time necessary is used. The configuration which solved the instance fastest is considered the winner and this information is then used to update an internal ranking procedure. When new instances arrives the ranking

produced by this ranking procedure is used to select which configurations to run. The current incumbent is always selected which provides a cap on the worst-case solving-time. The configuration pool is periodically refreshed by removing and replacing under performing configurations. Replacements are generated using a variety of techniques aimed at improving the overall quality of the pool.

1.2 Thesis Statement

The thesis defended in this dissertation is as follows:

Thesis. The performance of combinatorial solvers can be improved as a stream of instances is being processed without prior information or training. This is possible due to real-time algorithm configuration using parallel evaluation combined with a robust ranking system.

1.3 Thesis Contributions

This thesis makes a number of contributions to the existing body of knowledge:

- We introduce a variation of the algorithm configuration problem, *real-time algorithm configuration*, where the goal is to improve the configuration of a target algorithm while processing a stream of problem instances without increasing the time required to return a solution.
- We propose a framework outlining the key components required to solve the *real-time algorithm configuration* problem. This framework is called the Real-time Algorithm Configuration through Tournaments (ReACT) framework. It exploits the increasingly common parallel architectures available in order to determine improving configurations using a racing methodology and ranking approach.
- Two concrete instantiations of the ReACT framework are implemented. These demonstrate the flexibility of the framework and are used to empirically evaluate its effectiveness on a number of combinatorial optimisation benchmarks achieving performance on a par with or exceeding that of state-of-the-art offline configurators.
- We demonstrate that the ordering of the stream of incoming problem instances has an impact on the performance of the configurator. Further to this we show that the optimal selection method for choosing which configurations to run from pool of configurations is dependent related to this instance ordering.

- We introduce a novel method for reducing the cost of using genetic algorithms to optimise expensive functions by limiting the size of the competitions in each generation and using rank aggregation across generations to determine individual fitness.
- We show that feature selection techniques from the machine learning literature can be applied to algorithm configuration in order to reduce the configuration space and improve the search for good configurations.

1.4 Organisation of this Dissertation

The rest of the thesis is organised as follows:

In Chapter 2 we present a summary of the literature relevant to this dissertation. Specifically, we give an overview of combinatorial optimisation, in particular we outline the techniques, methods, and heuristics used to solve problems in the subfields of Constraint Satisfaction, Boolean Satisfiability, and Integer Programming. We also present a review of the literature related to algorithm selection and algorithm configuration.

Chapter 3 provides motivation for the research conducted in this thesis and defines the *real-time algorithm configuration* problem. We introduce the Real-time Algorithm Configuration through Tournaments Framework, ReACT, and discuss its constituent components.

Chapter 4 details two concrete instantiations of the ReACT framework. The design choices pursued for each of these implementations are defended and empirical evaluations on a number of benchmarks comparing these against the state-of-the-art are presented. The work presented Chapters 3 and 4 has appeared in the peer-reviewed publications:

> Tadhg Fitzgerald, Barry O'Sullivan, Yuri Malitsky, and Kevin Tierney. Online search algorithm configuration. In Carla E. Brodley and Peter Stone, editors, *Proceedings of the Twenty-Eighth AAAI Conference on Artificial Intelligence, July 27 -31, 2014, Québec City, Québec, Canada*, pages 3104–3105. AAAI Press, 2014.
>
> Tadhg Fitzgerald, Yuri Malitsky, Barry O'Sullivan, and Kevin Tierney. ReACT: Real-Time Algorithm Configuration through Tournaments. In Stefan Edelkamp and Roman Barták, editors, *Proceedings of the Seventh Annual Symposium on Combinatorial Search, SOCS 2014, Prague, Czech*

Republic, 15-17 August 2014. AAAI Press, 2014.

Tadhg Fitzgerald, Yuri Malitsky, and Barry O'Sullivan. ReACTR: Realtime Algorithm Configuration through Tournament Rankings. In Qiang Yang and Michael J. Wooldridge, editors, *Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence, IJCAI 2015, Buenos Aires, Argentina, July 25-31, 2015*, pages 304–310. AAAI Press, 2015.

The effect of instance ordering on the ReACT configuration procedure is shown in Chapter 5. We also compare the metrics used for selecting which candidates should compete in ReACT's tournaments and discuss the relationship between this and the instance ordering. The findings in this chapter have appeared in:

Tadhg Fitzgerald and Barry O'Sullivan. Analysing the effect of candidate selection and instance ordering in a realtime algorithm configuration system. In Ahmed Seffah, Birgit Penzenstadler, Carina Alves, and Xin Peng, editors, *Proceedings of the Symposium on Applied Computing, SAC 2017, Marrakech, Morocco, April 3-7, 2017*, pages 1003–1008. ACM, 2017.

Tadhg Fitzgerald and Barry O'Sullivan. Candidate selection and instance ordering for realtime algorithm configuration. *Fundam. Informaticae*, 166(2):141–166, 2019.

Chapter 6 reviews how best to maintain ReACT's configuration pool by removing underperforming configurations and how to generate quality replacement configurations. We examine a number of metrics that can be used to track the quality of a configuration and determine when it is sufficiently poor to be removed. Methods for generating new configurations using genetic algorithms and model-based feature reduction are also presented in this chapter.

Finally, in Chapter 7 we conclude and discuss some potential avenues for future work.

Chapter 2

Background

Summary. *This chapter details the relevant background required to understand the upcoming chapters. Firstly, we outline what combinatorial optimisation problems are, as well as the approaches and techniques employed to achieve solutions to these problems efficiently. The second section of this chapter details portfolio techniques for solving these problems and configuration methods used to fine-tune the solvers and solver portfolios. Finally, we look at some associated work in related fields of study so as the fully explore the topic of this dissertation. It is hoped that this section will provide the reader with all the background and intuition needed to read this dissertation as a stand-alone work.*

2.1 Combinatorial Problem Solving

Combinatorial optimisation techniques have been used to model and find feasible solutions for many practical applications such as software verification [DMB08], vehicle routing [TV14], data centre scheduling [DCMO16, CLN12], rostering [EJKS04], cutting-stock problems [PFCO15] in addition to a host of other challenging problem domains. Combinatorial optimisation involves finding an optimal feasible solution (either maximum or minimum) from a finite set of solutions. A feasible solution must satisfy any given constraints.

Combinatorial problems are considered amongst the most difficult to solve due to an exponential increase in problem difficulty as items are added. Take coin flipping as an example. Flipping a single coin results in two possible outcomes, heads or tails, two coins allows for 4 possible permutations, while three coins produces 8 possibilities. In

general, the number of potential outcomes is 2^n where n is the number of coins flipped and 2 is the branching factor (a coin flip can land heads or tails only). Combinatorial problems exhibit identical rapid exponential growth, often with a branching factor far in excess of two! This rapid growth relative to the number of variables in the problem is known as a combinatorial explosion.

Finding optimal solutions for these types of combinatorial problems requires exhaustive search which quickly becomes intractable as the problem grows. In algorithm complexity these problems are said to be in the **NP** (non-deterministic polynomial time) class. **NP** class problems have no "quick", polynomial time, solution unless $\mathbf{P} = \mathbf{NP}$. Though $\mathbf{P} \stackrel{?}{=} \mathbf{NP}$ is an open question in computer science, it is widely believed that $\mathbf{P} \neq \mathbf{NP}$ [Gas02, Gas12]. This implies that in the worst-case all combinatorial problems will take an exponential amount of time relative to their input size to solve.

Despite this discouraging forecast, combinatorial problems are extensively studied in a number of domains including operations research, constraint programming, artificial intelligence, mathematical optimisation, and others [BLP20, Wol98]. Powerful heuristics developed through years of research allow complete search methods to prune vast swathes of the search space rendering previously intractable problems solvable in an acceptable amount of time. These complete search methods are guaranteed to find a solution should one exist.

Although recent advances in the state of the art have improved the situation for complete solvers, certain difficult problems remain intractable due to their size or structure [CKT91, GW96, GMP+01]. In these cases it is necessary to adopt incomplete algorithms such as local search and genetic algorithms. These algorithms sacrifice completeness guarantees in favour of speed. Generally these algorithms run with a resource limit (time, steps, evaluations etc.) and terminate upon finding a solution or resource exhaustion. Incomplete algorithms have repeatedly been shown to perform well on certain problems (exceeding the performance of complete algorithms in many cases).

Due to the nature of **NP** problems there is no universal solution when it comes to selecting which algorithm or algorithm parameters to use. For this reason, algorithms often expose options to the end user such as which heuristics to use. The complexity of modern solving algorithms makes this a daunting task, even for experts in the field. Automatic algorithm configuration provides an easy method to configure algorithms well with minimal human interaction.

The rest of this section outlines various approaches to modelling combinatorial prob-

lems. It is hoped this will give the reader an intuition for how modern solving algorithms operate and what type of parameters are being configured in the upcoming sections of this dissertation. There are a number of common approaches to modelling combinatorial optimisation problems, each with there own strengths and weaknesses. Constraint programming (CP), outlined in Section 2.1.1, models problems as a series of variables with fixed domains and a set of constraints operating over these variables. Section 2.1.2 investigates the inference and search techniques used to resolve these types of problem. Boolean satisfiability (SAT) models seek a satisfying assignment for a Boolean formula. A formula consists of a set of Boolean literals (*TRUE*, *FALSE*) connected by means of logical operators. SAT problems can be thought of as a subset on constraint programming and so similar resolution techniques are employed. These problems are explored in more detail in Section 2.1.3. Finally, Section 2.1.4 provides an overview of integer linear programs (ILP). Integer linear programs are mathematical optimisation problems where some variables are restricted to integer values. These are solved using relaxations of the initial problem and search methods.

2.1.1 Constraint Satisfaction Problems

A constraint satisfaction problem (CSP) consists of a set of variables each of which can assume a value from a fixed domain, and a set of constraints which limit the values that variables are allowed take. A CSP solution assigns a single value to each variable such that no constraints are violated.

Definition 2.1.1. Formally, a CSP can be defined as three components V, D, and C [Mac77, RN16]:

V is the set of variables, $\{X_1, X_2, ..., X_n\}$.

D is the set of domains for each variable, $\{D_1, D_2, ..., D_n\}$. The values for each domain in D describe the permitted values that the corresponding variable in V can take on.

Finally, C is the set of constraints that specify the allowed combinations of values. A single constraint C_i consists of a $< scope, relationship >$ pair. The *scope* defines which variables the constraint applies to, while the *relationship* specifies what values they can take on.

Relationships can be in the form of extensionally specified sets of (dis)allowed tuples or as intensionally specified relations, such as *less than*, *greater than* and *not equal*. Constraints can apply to any number of variables in the CSP. Unary constraints operate on a single variable e.g. $X \neq 3$. A binary constraint works on a pair of variable e.g. $X = Y$. This generalises to any number of constraints, known as N-ary constraints.

Constraints which operate over an arbitrary number of variables are known as global constraints. For example "alldifferent" is a common constraint which enforces that the set of variables it is applied to are assigned distinct values. While it is often possible to achieve the same result using simpler constraints, global constraints often benefit by using highly optimised filtering algorithms. A full list of global constraints is available in the global constraint catalogue [BCR12].

Binary CSPs are often represented by a (primal) constraint graph. Vertices in the constraint graph represent variables. Edges in the constraint graph indicate constraints on the connected nodes. Constraint graphs allow for easier representation and reasoning about CSP problems.

Example 2.1.1. Figure 2.1 shows the constraint graph for a simple CSP consisting of three variables X, Y and Z each with the domains $1, 2, 3$ and the constraints $X < Y$, $Y < Z$, and $Z \neq X$.

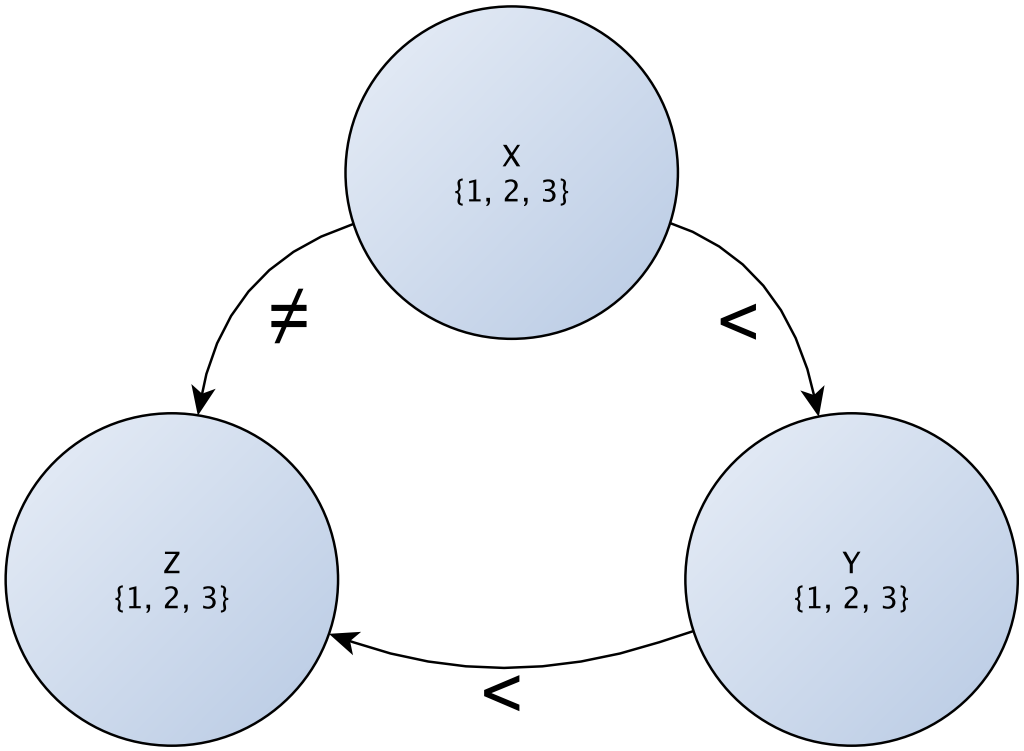


Figure 2.1: A simple constraint graph.

Constraint programming aims to allow users to model problems with ease [Bar99]. As Eugene Freuder eloquently stated "Constraint programming represents one of the closest approaches computer science has yet made to the Holy Grail of programming: the user states the problem, the computer solves it." [Fre97]. To this end constraint

programming languages and libraries provide a high-level method for abstracting the constraint problem model from the solving procedures. This form of declarative programming is similar to the way high-level programming languages abstract a program from the underlying systems architecture and assembly language. The CSP is defined using a constraint programming language or library such as MiniZinc [NSB+07a], Choco [PFL17] or Numberjack [HOO10]. This model is then solved using either an integrated or standalone backend solver, such as Mistral [Heb08], Lingeling [Bie10], or IBM CPLEX [IBM14], using advanced constraint optimisation techniques (these are discussed in depth in Section 2.1.2).

Example 2.1.2. A classic example of a problem well suited to solving by means of constraint programming is the N-queens problem [SS87]. This puzzle involves placing n chess Queens on an n x n chess board in such a way that conflict is avoided. Queens are in conflict if two Queens are placed on the same horizontal, vertical or diagonal squares. There are a number of ways of modeling this problem as a CSP with varying degrees of efficiency [Nad90]. One possible representation is to model the problem by applying "alldifferent" constraints to all rows, columns, and diagonals which limit the number of queens to at most one. Figure 2.2 shows both distinct solutions for the 4-queens problem. Figure 2.3 shows how to model the problem in MiniZinc [PJS20, NSB+07b].

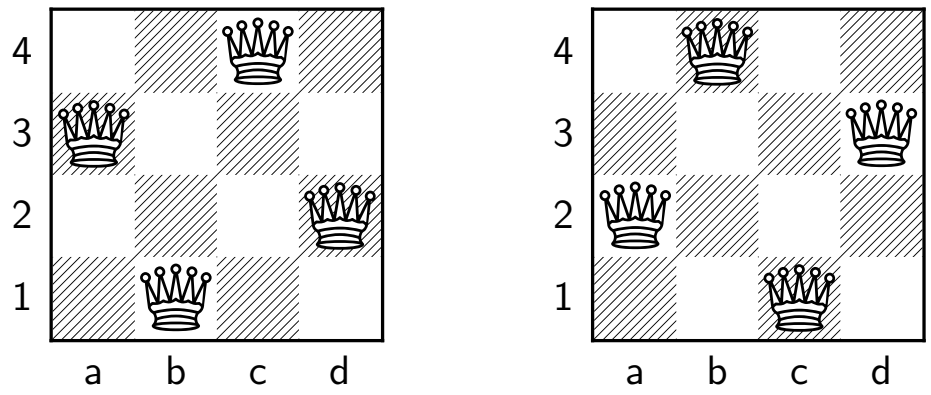


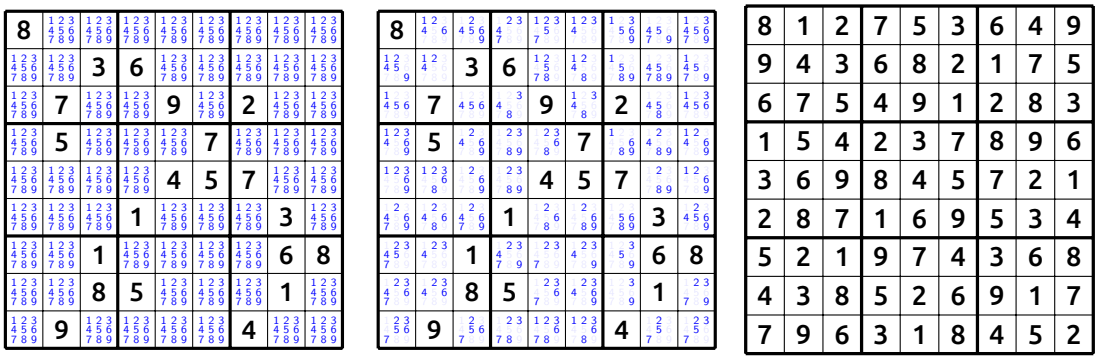
Figure 2.2: Both distinct solutions to the four queens problem.

Example 2.1.3. The logic puzzle Sudoku is another example of a problem which can be modeled and solved efficiently using constraint programming [Sim05]. Sudoku is a popular logic puzzle consisting of a partially completed 9×9 grid of integers. Each row, column and 3×3 region can only use the digits 1 to 9 once. The goal is to complete the remaining assignments. In this case each cell is a variable. The cells where a value i has been assigned have the domain $\{i\}$ while the domain for all unassigned

```
int: n;
% queen in column i is in row q[i]
array [1..n] of var 1..n: q;

include "alldifferent.mzn";

% distinct rows
constraint alldifferent(q);
% distinct diagonals
constraint alldifferent([ q[i] + i | i in 1..n]);
% upwards+downwards
constraint alldifferent([ q[i] - i | i in 1..n]);

% search
solve :: int_search(q, first_fail, indomain_min)
      satisfy;
output [ if fix(q[j]) == i then "Q" else "." endif ++
         if j == n then "\n" else "" endif | i,j in 1..n]
```

Figure 2.3: The N-Queens problem modelled in MiniZinc. Credit [PJS20, NSB+07b].

cells is $\{1, ..., 9\}$ as shown in Figure 2.4a. The problem can be modelled by applying "alldifferent" constraints to each row, column and square 3×3 region. Finally search over the reduced domains produces the final solution seen in Figure 2.4c. Figure 2.5 shows how this model can be written in MiniZinc [PJS20, NSB+07b]. The exact details of how these procedures are applied are discussed in the rest of this chapter.



(a) Sudoku domains given initial clues.

(b) "AllDifferent" constraints propagated over domains.

(c) A solution to the Sudoku puzzle.

Figure 2.4: Example of constraint propogation on the Sudoku puzzle. Credit [Hur16].

While the N-queens and Sudoku puzzles are simple examples that illustrate the power of CSP solvers there are many practical problems can also be modelled and solved using constraint programming. These include evacuation planning [ESVH15], call-centre scheduling [PRL+14], flight planning [KCL17], steel production [GS17], and wine

```
include "alldifferent.mzn";

int: S;
int: N = S * S;
% digits for output
int: digs = ceil(log(10.0,int2float(N)));

set of int: PuzzleRange = 1..N;
set of int: SubSquareRange = 1..S;

% initial board 0 = empty
array[1..N,1..N] of 0..N: start;
array[1..N,1..N] of var PuzzleRange: puzzle;

% fill initial board
constraint forall(i,j in PuzzleRange)(
    if start[i,j] > 0 then puzzle[i,j] = start[i,j]
     else true endif );

% All different in rows
constraint forall (i in PuzzleRange) (
              alldifferent( [ puzzle[i,j] | j in PuzzleRange ]) );

% All different in columns.
constraint forall (j in PuzzleRange) (
              alldifferent( [ puzzle[i,j] | i in PuzzleRange ]) );

% All different in sub-squares:
constraint forall (a, o in SubSquareRange)(
              alldifferent( [ puzzle[(a-1) *S + a1, (o-1)*S + o1] |
                                  a1, o1 in SubSquareRange ] ) );

solve satisfy;

output  [ show_int(digs,puzzle[i,j]) ++ " " ++
         if j mod S == 0 then " " else "" endif ++
         if j == N then
             if i != N then
                if i mod S == 0 then "\n\n" else "\n" endif
         else "" endif else "" endif
         | i,j in PuzzleRange ] ++ ["\n"];
```

Figure 2.5: The generalised Sudoku problem modelled in MiniZinc. Credit [PJS20, NSB+07b].

blending [VCT13] to name but a few.

2.1.2 Solving Techniques

Constraint problems are commonly solved using a combination of inference and search. Inference reduces the search space by ensuring that each value in a domain is supported by a value in other domains (consistency). Some problems can sometimes be solved by inference alone, but in most cases some form of search must be employed to find a feasible assignment. Search is often performed using a backtracking style search where each node in the search tree represents a variable assignment. However, due to the combinatorial nature of constraint problems, the worst case for backtracking search is an exhaustive search of the entire search tree. For this reason local search strategies are adopted for more complex problems.

2.1.2.1 Inference

Prior to search, inference and propagation are used to reduce the search space by ruling inconsistent values and simplifying the problem [Bes06]. Consistent values are those which can be shown to logically not conflict with the imposed constraints. There are different forms of consistency such as node-consistency, arc-consistency, and path-consistency which can deliver more or less filtering at the cost of extra computation.

Node-consistency enforces any unary constraints on the domain. This is trivially performed as a preprocessing step. For example a variable V with the domain $\{1, 2, 3\}$ and the constraint $V > 1$ can simply reduce the domain to $\{2, 3\}$. When node-consistency has been enforced the CSP is said to be domain consistent.

A stronger form of consistency is arc-consistency which applies to binary constraints. Given a pair of variables x and y with domains D_x and D_y respectively, and a constraint, $c_{xy} \in C$, operating on both variables. The variable x is arc-consistent with y if for all values in D_x there is a support in D_y such that a constraint c_{xy} is satisfied. A constraint is arc-consistent if all variables involved in the constraint, $c_{xy} \in C$ are arc-consistent with each other. A CSP is arc-consistent if all domains in the problem are arc-consistent.

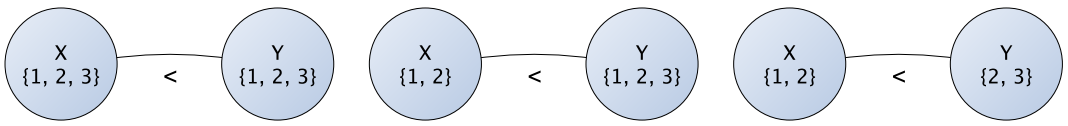
The most well known algorithm for enforcing arc consistency is the AC-3 algorithm as it is both simple and efficient [Mac77]. Algorithm 1 gives the pseudocode for the AC-3 algorithm. The REVISE function handles removing values which have no supports. REVISE is called by the AC-3 function which handles maintaining the list of arcs requiring consistency checks.

Example 2.1.4. To give a concrete example, Figure 2.6 shows constraint network for a

Algorithm 1 AC-3 Algorithm. Reproduced from Artificial Intelligence: A Modern Approach [RN16]

function AC-3(csp) *returns* $false$ if an inconsistency is found and $true$ otherwise
  inputs: csp, a binary CSP with components (X, D, C)
  local variables: $queue$, a queue of arcs, initially all the arcs in the csp
  while $queue$ is not empty **do**
    $(X_i, X_j) \leftarrow$ REMOVE-FIRST($queue$)
    if REVISE(csp, X_i, X_j) **then**
      if size of D_i = 0 **then return** $false$
      end if
      for all X_k in X_i.NEIGHBOURS - $\{X_j\}$ **do**
        add (X_k, X_i) to $queue$
      end for
    end if
  end while
  return $true$
end function
function REVISE(csp, X_i, X_j) **returns** $true$ iff we revise the domain of X_i
  $revised \leftarrow false$
  for all x in D_i **do**
    if no value y in D_i allows (x, y) to satisfy the constraint between X_i and X_j
then
      delete x from D_i
      $revised \leftarrow true$
    end if
  end for
  return $revised$
end function

CSP with two variables, X and Y, each with the domain $\{1, 2, 3\}$ (Figure 2.6a). The problem has a single constraint $X < Y$ (represented by an edge). As X must be strictly less than Y it is possible to remove the value 3 from the domain of X because there is no value in Y which is greater than 3 (Figure 2.6b). X is now arc-consistent with Y. Similarly, 1 can be removed from the domain of Y as there is no value in X which supports it (Figure 2.6c). Enforcing both of these makes the CSP arc-consistent.



(a) Initial domains. (b) Consistency on X. (c) Consistency on Y.

.

Figure 2.6: A constraint graph showing how arc consistency is achieved in Example 2.1.4

Arc-consistency can be extended so that it operates of n-ary constraints. This is called generalised arc consistency (GAC) [MM88, Bes06]. A variable x is generalised arc consistent with the constraint c if all values in the domain of x have support values in the domains of the other variables which c operates on. For example, given the variables $x \in \{1, 2, 3\}$, $y \in \{2, 3, 4\}$, and $z \in \{3, 4, 5\}$ and the constraint $x = y = z$. We can reduce all domains to $\{3\}$ using GAC as no other value is supported in all three domains.

While arc-consistency applies to single variables to make them consistent there is another form of consistency called path-consistency which considers pairs of variables (or constraints). A pair of variables, x and y, are path-consistent with a third variable, z, if for every value assignment (x_a, x_b) satisfying the constraints c_{xy} there is also a value in z such that the constraints c_{xz} and c_{yz} hold. Much like arc-consistency, there is a version of path-consistency that generalises to encompass an arbitrary number of variables rather than two.

Figure 2.4b shows the remaining values for variables in Example 2.1.3 after the "alldifferent" constraint has been propagated over the rows, columns and region. It is not possible to reduce the domains any further given the initial clues, so search is necessary in order to derive the final solution in Figure 2.4c. The various search methods and optimisations are discussed in Section 2.1.2.2.

Consistency checks are often interwoven with the search procedure during branching and backtracking in order to maintain a consistent state and reduce the search space as much as possible. As these checks occur repeatedly throughout search they are often limited to less computationally expensive forms of consistency such as arc consistency (in fact this procedure is known as maintaining arc-consistency). More computationally expensive procedures (such as path-consistency) are normally reserved to be used as a preprocessing procedure.

2.1.2.2 Backtracking Search

Inference and consistency checking can reduce the problem size greatly, sometimes even finding a solution (where each domain contains a single value) or proving infeasibility (where one or more domains are empty). However, often inference alone is not sufficient to find a solution a search routine is employed. Typically this is backtracking search where each node represents a value assignment to a variable [vB06]. Values are assigned until a inconsistency is detected at which time the search backtracks to assign another value. If search assigns all variables a value and no inconsistency is discovered we have a feasible solution.

Algorithm 2 Backtracking Search. Reproduced from Artificial Intelligence: A Modern Approach [RN16]

```
function BACKTRACKING-SEARCH(csp) returns a solution, or failure
    return BACKTRACK({}, csp)
end function
function BACKTRACK(assignment, csp) returns a solution, or failure
    if assignment is complete then return assignment
    end if
    var ← SELECT-UNASSIGNED-VARIABLE(csp)
    for value in ORDER-DOMAIN-VALUES(var, assignment, csp) do
        if value is consistent with assignment then
            {var = value} to assignment
            inferences ← INFERENCE(csp, var, value)
            if inferences ≠ failure then
                add inferences to assignment
                result ← BACKTRACK(assignment, csp)
                if result ≠ failure then return result
                end if
            end if
        end if
        remove {var = value} and inferences from assignment
    end for
    return failure
end function
```

The pseudocode for the basic backtracking search algorithm is shown in Algorithm 2. The function BACKTRACKING-SEARCH is simply a wrapper around the recursive function BACKTRACK which does the bulk of the work. BACKTRACK accepts a partial assignment of variables and the CSP problem as inputs. The function then tests value assignments until a feasible solution is discovered or an inconsistency is discovered (either by INFERENCE or a recursive BACKTRACK call). The INFERENCE call in this case can apply any of the forms of consistency discussed in Section 2.1.2.1, such as arc-consistency. In the case of inconsistency the variable assignment is undone and a different value trialled instead.

Basic backtracking search is guaranteed to find a solution if one exists but the time taken to discover the solution can vary dramatically. In the worst case an exhaustive search of all assignments is possible. To mitigate the chance of this a number of heuristics relating to how the search-tree is traversed have been developed. The variables in the search tree can be processed in any order (SELECT-UNASSIGNED-VARIABLE in Algorithm 2). This can have a large effect on the size of the search space and as such a number of variable ordering heuristics have been developed [Bré79, BHLS04,

GB65, DM94, BR96, HE80]. The most well-known of these is the "fail-first" principle which says that it is better to encounter variables with the smallest domain first [HE80]. The intuition here is that it is better to process variables which are likely to fail early rather than wasting time searching only to fail at the end. Another variable ordering which is sometimes used in conjunction with the "fail-first" heuristic is the degree heuristic [Bré79]. Here the most constrained variable is favoured as it prunes more of the search space than selecting variables which are less constrained.

The ORDER-DOMAIN-VALUES function allows BACKTRACK to enumerate the domain values in different ways. When selecting which variable to assign next we strive to fail fast, the opposite is true when choosing which value to assign. We endeavour to allow the maximum flexibility for future assignments by selecting the value which excludes the fewest future options. In constraint *satisfaction* problems search can cease and return success as soon as *any* satisfying assignment is found, so maximising the chances of finding a legal assignment is desirable. Again, a number of heuristics for value ordering have been proposed such as "min-conflicts", which looks at the sum of the remaining domains sizes, and "promise", which takes the product of the domain sizes [FD$^+$95, Gee92].

Example 2.1.5. Revisiting the previous 4-queens example (Example 2.1.2), Figure 2.7 shows the search tree generated using backtracking search to find a solution[1]. Initially we start with an empty 4×4 board. As there are four queens and four rows and columns, any solution must have a queen in each row and column so as not to be in conflict with one another. We denote the columns using the letters A-D and rows by the numbers 1-4. Search begins by placing a queen arbitrarily in A4 (the first available column from the left and row from the top). The red X's show the three new cardinality constraints that are propagated along the horizontal, vertical, and diagonal rows by this placement. Search proceeds by placing a queen in the next available square, B2. Again new constraint propagation is marked with red X's while previous constraints are marked in grey. This propagation leave only one square available with two queens left to place, so we must backtrack. Inference caused by the B2 placement is removed and the next available square, B1, is trialled. This allows a queen in only one position, C3, in column C. Placing a queen here eliminates the final possible square so backtracking occurs again. With no backtracking options available in column B or C, search backtracks to the initial assignment and changes this to A3. From this point on constraint propagation completely guides the search to a feasible solution.

[1] This example is inspired by the Google OR-Tools documentation [Dev18].

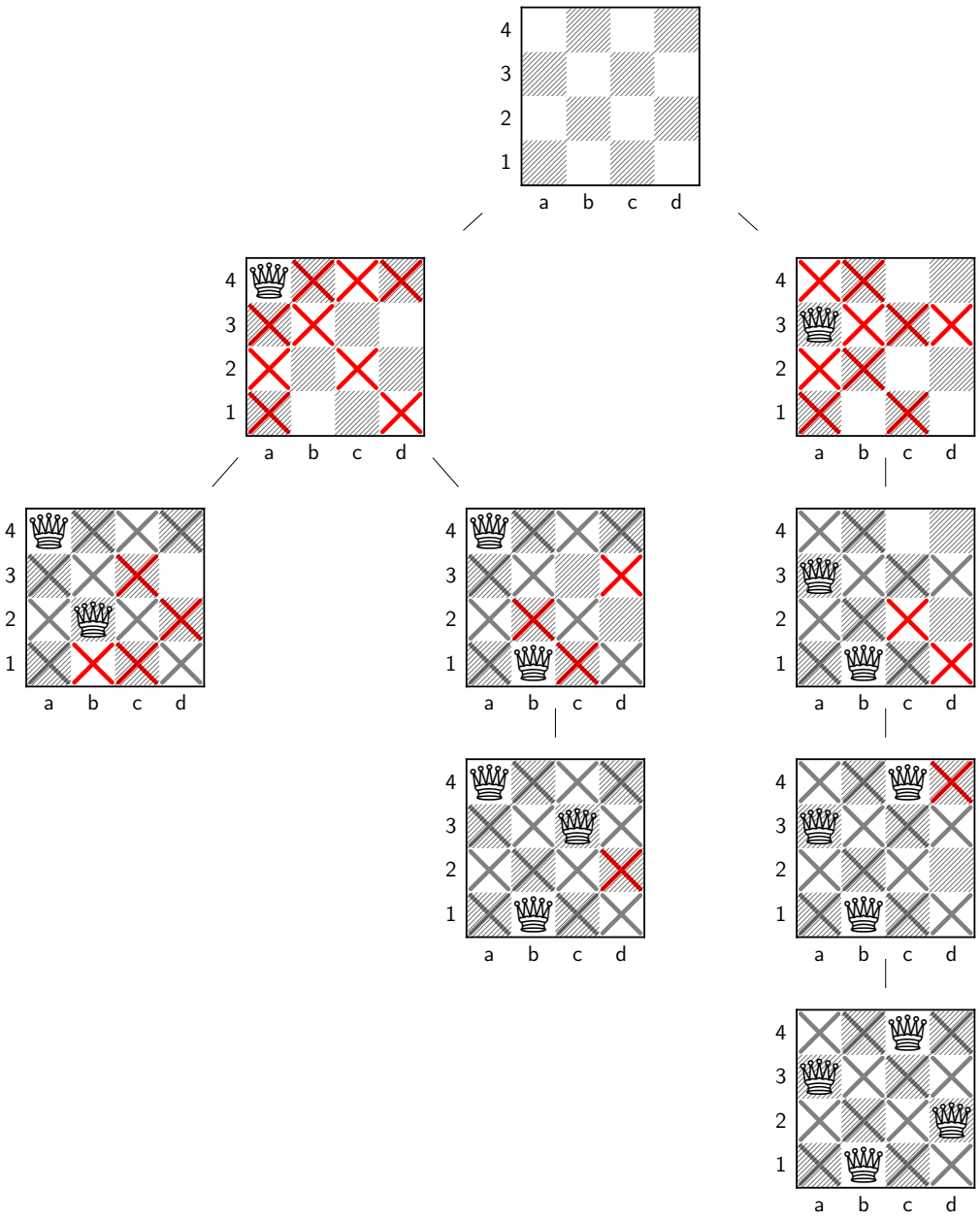


Figure 2.7: The search tree for the 4-queens problem.

2.1.2.3 Local Search

Backtracking search is complete which means it will discover a solution should one exist. This is certainly a desirable property but in the case of very large problems, consisting of many variables, exhaustive search proves impractical. For this reason, solutions to large problems are often sought using incomplete local search algorithms. Local search aims to find a solution which satisfies all constraints by iteratively improving on an initial assignment of variables [HT06]. This methodology is often able to find good or feasible solutions more quickly than complete algorithms but achieves this by sacrificing guarantees of optimality and completeness.

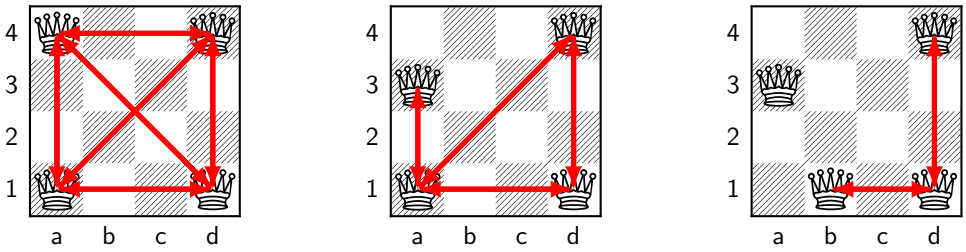
Hill-Climbing The simplest form of local search, hill-climbing, moves to neighbouring assignments by altering variable values in a greedy fashion. Which neighbour to move to is decided based on a cost function e.g. the number of conflicts.

Algorithm 3 Greedy Hill Climbing Local Search. Reproduced from Artificial Intelligence: A Modern Approach [RN16]

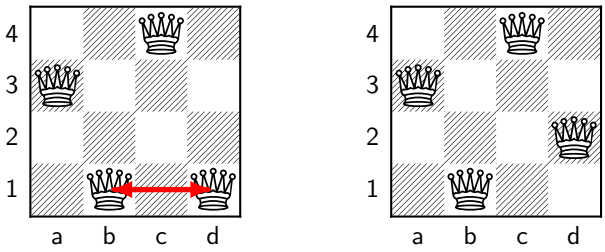
function MIN-CONFLICTS(csp, max_steps)
  $current \leftarrow$ an initial complete assignment for csp
  for i = 1 to max_steps **do**
    if $current$ is a solution for csp **then return** $current$
    end if
    $var \leftarrow$ a randomly chosen conflicted variable from csp.VARIABLES
    $value \leftarrow$ the value v for var that minimises CONFLICTS(var, v, $current$, csp)
    set var = $value$ in $current$
  end for
  return $failure$
end function

Algorithm 3 outlines the pseudocode for a hill-climbing algorithms that aims to minimise the number of conflicts in a CSP [MJPL92]. The algorithm greedily selects a variable associated with a violated constraint at random then alters its value such that the number of conflicts in the CSP is minimised. This process is repeated until a feasible solution is discovered or a predefined resource limit, max_steps, is exhausted. Hill-climbing local search is a simple algorithm that illustrates the idea nicely but is rarely used in practice as it is very prone to becoming trapped in local optima.

Example 2.1.6. Taking the 4 queens problem used previously as a concrete example, Figure 2.8 shows the updates made by hill-climbing local search to discover a solution. In contrast to backtracking search hill-climbing starts with a complete initial assignment that violates some of the constraints. Figure 2.8(a) shows that a queen has been placed



(a) 6 conflicts (b) 4 conflicts (c) 2 conflicts



(d) 1 conflict (e) The solution

Figure 2.8: Local search on the 4 queens problem.

in each corner of the chequerboard leading to 6 conflicts in total (conflicts are shown with red arrows). From this initial assignment future placements are calculated by greedily selecting the queen placement that minimises the number of conflicts from a "neighbourhood" of possible assignments. This "neighbourhood" can be as small as the neighbouring cells for each queen or as large as all possible legal placements. In Algorithm 3 the function CONFLICTS decides the neighbourhood and computing the number of conflicts for each neighbour. By following the a trail of improving assignments, Figures 2.8(b-d), we eventually arrive at a solution in Figure 2.8(e). In this particular example we arrived at a solution, however in many cases simple hill-climbing becomes stuck in local optima. The reason for this is that hill-climbing search will not accept non-improving moves which are often necessary to escape local optima. One slight adaptation of greedy hill-climbing allows for moves to neighbours which have an equal (but not worse) objective value to the current assignment. This variant is known as plateau search and can lead to marked improvements in search performance [HK93]

Tabu Search One classical improvement on hill-climbing is tabu search [Glo89, Glo90b]. Two of the major drawbacks of basic hill-climbing local search are its tendency to become trapped in local optima, and to cycle when plateaus with states of equal objective value are encountered. Tabu search addresses both of these issues by

allowing disimproving moves within a neighbourhood and by maintaining a tabu list. A tabu list is a list of previously visited states that the search is prohibited from revisiting. The tabu list serves to drive the search towards new parts of the state space and avoids cycling. The search visits the best neighbouring solution that is not on the tabu list even if this neighbouring state has a worse objective value, in this way local optima can be escaped. In order to remain memory efficient the length of the tabu list is often set to a fixed length or dynamically adjusted. In addition to the short term memory used to prevent cycling, tabu search also adopts intermediate and long term memory structures which aim to drive intensification and diversification respectively [Glo89, Glo90a].

Random Walk Stochasticity can also be used to avoid the pitfalls of greedy search. One method is to adopt a random walk approach where greedy search is interspersed with random non-greedy assignments with a certain probability. This leads to a semi-random walk over the possible assignments in the search space. By occasionally making random assignments rather than following a purely greedy strategy local optima can be escaped. This idea was initially outlined in the WalkSAT algorithm for Boolean satisfiability problems but can easily be adapted to constraint satisfaction problems [SK93, SKC94, DC03, Sch99].

Constraint Weighting Another method of escaping local minima is to alter the objective function as search progresses. Constraint weighting or breakout methods achieve this by weighting constraints which are frequently violated more heavily [Mor93]. Such methods are easily integrated as part of other search procedures such as random walks [SK93]. This changes the reward surface and in doing so encourages the search to correctly assign values to satisfy the more difficult constraints. Formally the objective function for breakout search is given as $F(\bar{a}) = \sum_i w_i * C_i(\bar{a})$ [DC03]. The current weight of the constraint C_i is given by w_i, while $C_i(\bar{a})$ is an indicator variable set to 1 if the constraint is violated and 0 otherwise. When a local minima is encountered all weights associated with the currently violated constraints are incremented. This changes the objective function and allows search to proceed past the previous local minimum.

Simulated Annealing There are also more sophisticated stochastic algorithms, such as simulated annealing that uses a cooling schedule to dynamically control the ratio of random and greedy movements [KGV83]. Simulated annealing is inspired by the annealing process in metallurgy where properties of the metal are dictated by how it is heated and cooled. Similarly simulated annealing has a cooling schedule which reduces the likelihood of accepting worsening moves over time. Algorithm 4 outlines

the pseudocode for the SIMULATED-ANNEALING algorithm. The algorithm starts with an initial random assignment, $current$. At each time step a neighbour is chosen by altering variables' values randomly, $neighbour$. If the objective value of $neighbour$ (in this case the number of constraints violated, CONFLICTS) improves over $current$, $neighbour$ becomes the new incumbent. If $neighbour$'s objective value is worse then it is accepted with a certain probability which is dependant on the cooling schedule. A solution is returned if an assignment with no conflicts is found otherwise a partial assignment is returned once the "temperature", T, reaches 0. In this way simulated annealing accepts more worsening moves initially allowing it to explore a larger portion of the search space. Later in the search, as the temperature converges towards 0, mostly improving assignments are accepted similar to hill-climbing.

Algorithm 4 Simulated Annealing Local Search. Adapted from Artificial Intelligence: A Modern Approach [RN16]

function SIMULATED-ANNEALING(csp, $schedule$) **returns** a solution state
  inputs: csp, a constraint satisfaction problem
    $schedule$, a mapping from time to "temperature"
  $current \leftarrow csp$.INITIAL-ASSIGNMENT
  for t = 1 to ∞ **do**
    $T \leftarrow schedule(t)$
    if T = 0 **or** $current$.CONFLICTS **then return** $current$
    end if
    $next \leftarrow$ a randomly selected neighbour of $current$
    $\Delta E \leftarrow next$.CONFLICTS - $current$.CONFLICTS
    if ΔE > 0 **then** $current \leftarrow next$
    else $current \leftarrow next$ only with probability $e^{\Delta E/T}$
    end if
  end for
end function

2.1.2.4 Constraint Optimisation

Up to this point we have looked at constraint *satisfaction* problems where the focus is on finding any feasible solution. Another, closely related, class of problems called Constraint *Optimisation* Problems (COPs) add a cost function to minimise or an objective function to maximise [DC03, FW92]. Searching for an optimal solution in the space of possible solutions is a far more difficult than the constraint satisfaction problem as it is not possible to finish once *any* solution is found.

Combinatorial auctions are one example of a COP. Combinatorial auctions differ from conventional auctions in that bidders bid on different bundles of items as opposed to a single item. The optimisation problem arises in trying to find a set of bids which

maximises profit for the auctioneer while respecting the constraint that no item can be sold twice. Such auctions arise frequently in areas such as wireless spectrum auctions, airport time slots, and network routing, among others [DVV03].

Similarly, Weighted Constraint Satisfaction Problems (WCSPs) allow certain soft constraints to be violated in order to find an approximate solution to the given problem. Each constraint is weighted according to its importance in a solution. The goal then is simply to maximise the sum of the weights of the satisfied constraints.

2.1.3 Boolean Satisfiability

Description and Examples An important subset of CSPs is the Boolean satisfiability problem (SAT). The Boolean satisfiability problem entails finding an assignment to a set of Boolean literals (*True, False*) that satisfy a propositional logic formula if such an assignment exists or determining unsatisfiability otherwise. The SAT problem is well known as it was the first to problem shown to be NP-complete [Coo71]. The problem is quite significant as all other NP-complete problems can be transformed to SAT and is at the heart of the $P \stackrel{?}{=} NP$ question. SAT also has many practical applications, in particular it is used heavily in circuit design. As a testament to the importance of SAT in computer science, SAT competitions to find the best solvers have been running for over twenty years [BHJ17, BH$^+$16, BDHJ14, BBH$^+$13, SLBH05, LBS04, JLBRS12, LBS03].

SAT problems consist of Boolean variables and the Boolean operators *and* ($\wedge$), *or* ($\vee$), and *not* ($\neg$). A literal is a Boolean variable or its negation. Problems that have a satisfying assignment are called *satisfiable*, while those which do not are *unsatisfiable*. Normally, SAT problems are written in conjunctive normal form (CNF) where the expression is a conjunction of clauses, and each clause is a disjunction of literals (an *and* of *ors*). CNF formulas represent the SAT problem nicely in that a formula is satisfiable if every clause is satisfied, and a clause is satisfied if any literal is assigned a *true* value. All propositional formulas can be converted to CNF easily. The below expression is an example of a SAT formula which can be satisfied by assigning $x_1 = True, x_2 = True$:

$$(x_1 \vee x_2) \wedge (x_2 \vee x_3) \wedge (x_1 \vee \neg x_3).$$

DPLL Similar to CSP problems, backtracking search is at the core of most complete SAT solvers. Many powerful SAT solvers are based on the Davis–Putnam–Logemann–Loveland (DPLL) algorithm [DP60, DLL62]. DPLL recursively tests assignments of Boolean values to literals in the CNF formula. *Unit*

propagation and *pure literal elimination* form the basis of the recursive DPLL algorithm which is outlined in Algorithm 5.

Algorithm 5 DPLL Algorithm. Reproduced from Artificial Intelligence: A Modern Approach [RN16]

function DPLL-SATISFIABLE?(s) **returns** $true$ or $false$
  $clauses \leftarrow$ the set of clauses in the CNF representation of s
  $symbols \leftarrow$ a list of the proposition symbols in s
  return DPLL($clauses$, $symbols$, {})
end function
function DPLL($clauses$, $symbols$, $model$) **returns** $true$ or $false$
  if every clause in $clauses$ is true in $model$ **then return** $true$
  end if
  if some clause in $clauses$ is false in $model$ **then return** $false$
  end if
  P, $value \leftarrow$ FIND-PURE-SYMBOL($symbols$, $clauses$, $model$)
  if P is non-null **then return** DPLL($clauses$, $symbols - P$, $model \cup \{P = value\}$)
  end if
  P, $value \leftarrow$ FIND-UNIT-CLAUSE($clauses$, $model$)
  if P is non-null **then return** DPLL($clauses$, $symbols - P$, $model \cup \{P = value\}$)
  end if
  $P \leftarrow$ FIRST(symbols); $rest \leftarrow$ REST($symbols$)
  return DPLL($clauses$, $rest$, $model \cup \{P = true\}$) **or** DPLL($clauses$, $rest$, $model \cup \{P = false\}$)
end function

A unit clause is a clause where all bar one of the literals have been assigned a value of *false*. As CNF is a conjunction of disjunctions, each clause must have at least one *true* assignment for an formula to be satisfied. This trivially allows the literal to be assigned and removed from consideration. In Algorithm 5 FIND-PURE-SYMBOL identifies pure literals which are then assigned. As an example of pure literal elimination, take the clause $(\neg A \vee B \vee C)$ and the assignment $A = True, B = False, C =?$. C is a unit clause as the assignments to both the literals $\neg A$ and B evaluate to*false*. Given the current values the only satisfying assignment for C is *true*. Often fixing a single literal as the result of a unit clause results in a cascade of assignments known as unit propagation.

For example, imagine the clause from the previous example formed part of the larger expression $(\neg A \vee B \vee C) \wedge (\neg C \vee \neg D) \wedge (D \vee E)$. We know $C = True$ but this also forces $D = False$ which in turn implies $E = True$. These types of chain reactions prune vast swaths of the search tree and allow the DPLL to search the space more

efficiently.

In cases where a literal is the same in all clauses (i.e. always negated or always without negation), it is possible to assign that literal a truth value which satisfies all clauses it is a part of. For example, in the formula $(\neg A \vee B) \wedge (B \vee C)$, B can be assigned a value of *true* in order to satisfy the assignment. Once the literal has been assigned a fixed value all clauses which it is a part of become satisfied, it is then possible to simplify the expression by removing these clauses.

Conflict-Driven Clause Learning DPLL techniques outlined previously form the backbone of many powerful SAT solvers. However, modern solvers which are built to tackle problems containing thousands of variables and millions of clauses must adopt additional enhancements such as variable and value ordering (similiar to that seen in CSPs), and most notably conflict-driven clause learning [SS96, BJS97, BHvMW09].

Conflict-driven clause learning (CDCL) extends the DPLL algorithm by learning clauses and introducing non-chronological backjumping. Similar to DPLL, CDCL arbitrarily selects values for variables which cannot be inferred and applies unit propagation. In addition to this CDCL builds an implication graph which tracks which assignments caused certain selections. When a conflict occurs analysing this implication graph allows the SAT solver to determine the assignments which lead to the conflict and create a new clause expressly forbidding those. This clause learning is a way of inferring additional useful clauses not explicitly stated in the original formula. As the resulting expression is tighter than the original it is possible to reduce the search space and amount of backtracking required dramatically. In addition to tightening to problem CDCL uses non-chronological backtracking where instead of backtracking to a parent the search backjumps to the first assignment in the new learnt clause. While CDCL was first introduced in the GRASP SAT solver [SS96] it is widely used in many successful SAT solvers today including Chaff [MMZ$^+$01], MiniSat [ES03], Clasp [GKNS07], Glucose [AS09], and Lingeling [Bie10], to name but a few.

Local Search Complete backtracking search procedures in SAT, despite aggressive pruning of the search tree using unit propagation and learned clauses, suffer from the same limitations encountered previously with CSPs, namely that guaranteeing completeness requires exhaustive enumeration of the search tree in the worst case. The solution again relies upon incomplete search algorithms similar to those used for CSPs.

One approach which have been particularly successful in the past are methods based on "random walks" such as GSAT and WalkSAT [SLM$^+$92, SKC94]. Algorithm 6 outlines

the WalkSAT algorithm. First the algorithm selects a random clause from the unsatisfied clauses. Then with a certain probability flips the value of a randomly selected literal in clause, otherwise it chooses the literal which will maximise the number of satisfied clauses. By approaching the problem in this manner WalkSAT avoid becoming trapped in local optima.

Algorithm 6 WalkSAT Algorithm. Reproduced from Artificial Intelligence: A Modern Approach [RN16]

function WALKSAT($clauses$, p, max_flips) *returns* a satisfying model or $failure$
  inputs: $clauses$, a set of clauses in propositional logic
  p, the probability of choosing to do a "random walk" move, typically around 0.5
  max_flips, number of flips allowed before giving up
  $model \leftarrow$ a random assignment or $true/false$ to the symbols in $clauses$
  for i = 1 to max_flips **do**
    if $model$ satisfies $clauses$ **then return** $model$
    end if
    $clause \leftarrow$ a randomly selected clause from $clauses$ that is false in $model$
    with probability p flip the value in $model$ of a randomly selected symbol from $clause$
    else flip whichever symbol in $clause$ maximises the number of satisfied clauses
  end for
  return $failure$
end function

2.1.4 Integer Programming

Integer (linear) Programming (ILP) problems are a class of mathematical optimisation problems similar to Linear Porgramming (LP) problems with the added restriction that all of the variables must adopt discrete (integer) values [Wol98]. Like LP problems the goal is to minimise (or maximise) some objective value subject to the linear constraints imposed on the problem. Often the integer restriction is required for practical reasons e.g. in a airline scheduling problem you cannot schedule half a person to work on a plane. A variation of the problem, known as Mixed Integer Programming (MIP) relaxes this requirement by allowing some real valued variables. Formally, a MIP problem is described as [Wol98]:

$$\min cx + hy \tag{2.1}$$

$$Ax + Gy \leq b \tag{2.2}$$

$$x \in \mathbb{R}_+, y \in \mathbb{Z}_+$$

The cost function to minimise is specified in 2.2. Here, x and y are positive real-valued and integer-valued vectors of length p and n respectively which represent the decision variables to optimise. c and h are p-dimensional and n-dimensional row vectors respectively known as the objective coefficients. Equation 2.1.4 specifies the (in)equalities to be satisfied in the MIP. b is a vector of length m known as the requirements. A and G are $m \times p$ and $m \times n$ matrices.

Mixed integer programming is used in a wide variety of practical applications such as production planning [PW06], cancer treatment [LFC03], crew scheduling [Wed95], combinatorial auctions [ATY00, Nis00] and many others. Example 2.1.7 gives an example of a combinatorial auction winner determination problem formulated as an integer program.

Example 2.1.7. A simple integer programming model of a combinatorial auction is as follows [Nis00]:

$$\max \sum_{i=1}^{n} x_i p_i$$

$$\text{s.t.} \sum_{i|j \in S_i} x_i \leq 1, \text{ for each } j = 1..m.$$

$$x_i \in \{0, 1\} \text{ for each } i = 1..n.$$

Here we have a set of bids indexed by i with n bids in total, $\{B_i|1..n\}$. A single bid, B_i consists of a mapping of item bundles, S_i, to prices p_i, $B_i = (S_i, p_i)$. There are m individual items, $\{g_1...g_m\}$ in the set G. The decision variable x_i indicates whether B_i is a winning bid.

To make the example concrete, lets consider a case with three bids, B_1, B_2, B_3, and four items, $\{g_1, ..., g_4\}$.

The bids are as follows $B_1 = (\{g_2, g_4\}, 3), B_2 = (\{g_1, g_2, g_3, g_4\}, 5), B_3 = (\{g_1, g_3\}, 3)$. This problem can then be formulated as:

$$\max x_1 3 + x_2 5 + x_3 3$$
$$\text{s.t. } x_1 + x_2 \leq 1$$
$$x_2 + x_3 \leq 1$$
$$x_1, x_2, x_3 \in \{0, 1\}$$

The objective coefficients represent the value of each bid while the inequalities prevent bids with overlapping items from being accepted. In this case the optimal assignment is $x_1 = 1, x_2 = 0, x_3 = 1$.

While the solution to the toy example above is trivial to solve manually, real world examples can often feature thousands of items and bids [DVV03]. Large challenging problems such as these require advanced solvers such as IBM ILOG CPLEX [IBM14]. Such solvers adopt a number of advanced heuristics and optimisations. Similar to solvers for other combinatorial optimisation problems search forms the basis for many powerful MIP solvers.

In particular a technique known as branch and bound is commonly used [MJSS16]. Branch and bound solves the easier linear relaxation of the problem (without integer constraints) using standard linear programming techniques such as the simplex method [Dan65]. Should this method provide an integer solution the algorithm can return a result and terminate. Otherwise, the problem is then split (branching) by altering a fractional decision variable such that it is no longer in violation of the integer constraints (rounding up and down). For example if $x = 7.5$ is a decision variable in the linear relaxation, two branches are created: one where $x \leq 7$ is a constraint and another with the constraint $x \geq 8$. This produces two sub-problems which are again solved by linear programming methods. In this way the problem is solved by recursively enumerating solutions.

Such enumeration would prove far too costly without also limiting to the number of states to evaluate through bounding. The linear relaxation provides an upper bound on the best objective value as a the relaxation of the problem will always be as good or better than the constrained problem. A lower bound can also be calculated by rounding all non-integer variables. As branches are explored the bounds can be tightened which reduces the search effort significantly and provides an indication of how close to optimal the current solution is, the so called optimality gap. Branches which are not explored are said to be fathomed or pruned [MJSS16]. Fathoming occurs for three reasons; an optimal solution has been found (fathomed by optimality), the solution is infeasible

(fathomed by infeasibility), or the objective value of the current LP solution is worse than the objective value of the current incumbent (fathomed by bounds).

Akin to what we've seen with other combinatorial optimisation solvers, the order that the decision variables are processed in influences how effectively the bounds can be tightened and as a result the amount of pruning that can be applied. There are numerous ordering heuristics such as lexicographic ordering, selecting the variable with the largest fractional value, strong branching, and pseudo costs [MJSS16]. Strong branching and pseudo costs in particular are very effective and used in many powerful solvers. Strong branching partially solves the LP to evaluate branching strength [AKM05]. The pseudo-costs heuristic maintains counters for each variable which are updated as search progresses to indicate which variables have historically been good to branch on [BGG$^+$71].

Cutting planes are another common technique used to resolve mixed integer programming problems. These are used in conjunction with branch and bound search (sometimes called branch and cut). There are a number of different cutting planes methods such as Gomory cuts [Gom60, Gom63] and lift-and-project cuts [BCC93]. While the methods are different in general all work in a similar manner by generating additional inequalities on the fly in order to prohibit infeasible solutions from future LP solutions.

2.2 Algorithm Configuration and Selection

Today there is an ever growing variety of algorithms being developed for solving hard combinatorial problems. Very often individual algorithms will perform well on a particular type of instances but experience less success more generally. Because of this it is often advantageous to create a portfolio of complementary solvers and select which to use on a per-instance basis. Similarly, for advanced solvers which expose a large number of parameters to control their behaviour there is often no single configuration which will perform optimally across all instances. Even if there was, determining this configuration by manual ad hoc experimentation would be near impossible. Because of this the related fields of algorithm configuration and algorithm selection have received a large amount of research attention in recent years [Smi08, Kot14, KHNT19, Hoo12b]. In the rest of this section we outline the various proposed approaches to these problems and review the relevant related literature.

2.2.1 Algorithm Configuration

Choosing the correct values for an algorithm's parameters can greatly increase the algorithm's performance over the default settings, sometimes by orders of magnitude [HHLBS09]. For this reason a variety of different methods have been proposed to tackle the automatic algorithm configuration problem which we will review in this section.

Problem Definition To begin we formally, define the algorithm configuration problem as follows [Hoo12b]: Given

- an algorithm A with parameters $p_1, ..., p_k$ that affect its behaviour,
- a space C of configurations (i.e. parameter settings), where each configuration $c \in C$ specifies values for A's parameters such that A's behaviour on a given problem instance is completely specified (up to a possible randomisation of A)
- a set of problem instances I,
- a performance metric m that measures the performance of A on instance set I for a given configuration c,

find a configuration $c^+ \in C$ that results in optimal performance of A on I according to metric m.

There are a number of different types of parameters to consider depending on the context where algorithm configuration is applied:

- Categorical: A finite, discrete set of unordered values.
- Numerical: Either integer or real valued parameter values.
- Ordinal: A finite discrete set of ordered values.
- Conditional: Certain parameters which are only activated when other parameters assume a certain value.

Similarly, the performance metric adopted varies depending on context, however the two most commonly used metrics optimised are runtime minimisation and solution quality.

2.2.1.1 Offline Configuration

The most common automatic algorithm configuration paradigm at the moment is offline algorithm configuration. Here, a training set of problem instances are collected which

the configurator uses to learn a suitable configuration during an offline training phase. At the end of the training phase a configuration is returned which is subsequently used to solve all future problem instances during the production phase. For these methods to be effective it is imperative that the training set is large enough, as too few examples will likely produce a configuration which overfits the training data. It is also a necessary condition of these methods that the training set be representative of the types of instances encountered during the production phase.

Early Parameter Tuning Early attempts at automated algorithm configuration for the field of constraint optimisation began in the early 2000's. During this time a number of different approaches to the parameter tuning problem were presented using techniques such as gradient-free numerical optimisation methods [HO01, AO06], and experimental design based methods [BSPV02, CGRW01, AL06] (see [ES11] for a broader overview). These first approaches were typically limited to optimising only a small number of continuous parameters [HO01, BSPV02, AL06, AO06]. For this reason, Hoos refers to these early approaches as *parameter tuning* and reserves the term *algorithm configuration* for "target algorithms with many categorical parameters...dealing with an algorithm schema that contains a number of instantiable components (typically, subprocedures or functions)" [Hoo12b].

CALIBRA The CALIBRA system, introduced in 2006, uses both experimental design techniques alongside a local search procedure in order to find improved configurations [AL06]. The algorithm uses Taguchi fractional factorial experimental designs to evaluate each parameter for two levels (i.e. two parameter values). The results of these evaluations are used to identify promising areas of the configuration space which are then explored by local search. The configurator identifies three levels (values) per parameter using the initial evaluations then uses these in combination with fractional experiment designs to identify nine promising configurations for each local search iteration to explore. Local search continues to iterate and refine the values until a local optimum is reached. At this point the search is restarted and three new values per parameter are identified using the current incumbents. These values are again used to identify neighbours for the local search using a fractional experiment design. This cycle continues until the number of evaluations exceeds a predefined threshold.

CALIBRA showed that it was able match or exceed the performance of hand-tuned solvers without any prior information. Despite this, the system is severely handicapped due to the fact that it is limited to configuring a maximum of only five parameters, and it's ability to only configure parameters in the continuous domain.

F-Race and Iterated F-Race Another early algorithm configuration approach, F-Race, adopts a racing approach [BSPV02]. Here a fixed set of configurations compete against one another until sufficient statistical evidence is gathered to eliminate inferior configurations. The pool of configurations is either provided by the end user or sampled using a full factorial design. This sampling methodology makes F-Race, as it was originally proposed, unsuitable for configuring anything but a small number parameters. Problem instances are solved one at a time by all configurations, after each iteration, a non-parametric Friedman test is applied to determine if any configurations are statistically inferior. These under-performing configurations are then removed and future races run with only the remaining configurations. This continues until a single configuration remains or the configuration budget is exhausted.

An extension of F-Race which iteratively runs races allows parameter optimisation over larger configuration spaces [BBS07, BYBS10]. The Iterated F-Race extension splits the total configuration budget into a number of iterations. During each iteration the F-Race procedure is run in the normal way. Initially, Iterated F-Race uses random sampling to provide the initial set of configurations which avoids the combinatorial explosion encountered using previous the full factorial design. While the initial sampling is performed uniformly at random, later iterations replenish the pool of configurations using a probability model which biases the selection using information from previous iterations elite configurations. In this way, the starting configuration pool for each iteration becomes successively stronger by refining the probability model and biasing the search towards good configurations.

Development of Iterative F-Race is continuing in the form of the irace package [LIDLC$^+$16]. This package implements a number of improvements, such as soft-restarts and elitist racing, not discussed in the original papers. Soft-restarts ensures diversity by expanding the range of allowed configurations when the probability model has constrained the generation procedure too much and caused new configurations to be near identical to previous configurations. Elitist iterated racing prevents accidental removal of configurations which have exhibited strong historical performance. The standard version of Iterated F-Race performs the statistical test and removal based only on runs in the current iteration. This can lead to situations where an unlucky sequence of instances can cause the removal of a strong configuration. Elitist racing combats this by ensuring that an elite configuration can only be removed after the configuration dominating it has run on the same number of configurations *overall*.

F-Race and Iterated F-Race were originally designed for scenarios where the goal was to optimise solution quality within a fixed time rather than reduce the overall runtime.

Due to this the configurators under perform relative to other algorithm configurators in runtime minimisation scenarios due to not limiting the time spent evaluating weak configurations. An extension introducing an adaptive capping mechanism, similar to that in ParamILS, was proposed in 2017 [CLIHS17]. This extension computed runtime bounds using the runtime performance of elite configurations and used these to terminate clearly inferior configurations early. This resulted in performance which was comparable to the state of the art black box configurators in runtime minimisation scenarios.

ParamILS The ParamILS automatic configuration framework, introduced in 2007, is able to configure arbitrary algorithms with very large numbers of discrete parameters [HHS07, HHLBS09]. The framework uses a local search heuristic to explore the configuration space. Specifically, ParamILS performs iterative first improvement search using a one-exchange neighbourhood (changing a single parameter at a time). The iterated local search generally begins with the default configuration as the incumbent and proceeds in three stages. Firstly, the a new candidate configuration is created by randomly assigning a set number of parameters from the configuration space. Then, first improvement local search is performed by exploring each neighbourhood which differs from the candidate by at most one parameter. The candidate is updated as soon as an improvement is discovered and the local search begins anew from the improved configuration. This process continues until there is no neighbouring configuration can improve on the candidate, a local optimum. The final stage uses an acceptance criterion to compare the locally optimal candidate and the current incumbent. If the candidate is accepted it becomes the new incumbent and the cycle continues. In order to avoid becoming trapped in local optima the iterated local search also uses random restarts. Here, with a certain probability the search jumps to another random part of the configuration space. On the termination of the search the configuration with the best performance is returned.

The paper introduces two concrete instantiations of the ParamILS framework, BasicILS and FocusedILS. The two methods differ in the method used to evaluate whether one configuration is better than another, namely the number of instances evaluated. BasicILS runs both configurations on the same fixed size set of instances (and seeds) and deems the configuration with better performance (e.g. mean runtime) over the set to be superior. FocusedILS on the other hand adaptively selects the number of instances to evaluate two configurations on. FocusedILS will only consider the performance of one configuration better than another if they have both run on the same number of instances. The configuration with a smaller number of runs performs additional

runs until both have the same number of runs. In the case where both configurations have the same number of runs one extra run is performed for each. Additionally the FocusedILS comparison procedure tracks the number of configurations evaluated since the last improvement and performs this many additional runs with a new incumbent when it is discovered. This ensures that good configurations have a large number of numbers and so lowers the likelihood of incorrectly removing a strong configuration.

Adaptive capping was introduced in a follow up journal paper in 2009 [HHLBS09]. This addresses the issue that if solvers are allowed to execute to completion much of the configuration time is spent evaluating sub par configurations. Adaptive capping remedies this by terminating runs whose execution time exceeds a bound set by the current best configuration. The paper proposes two versions of this mechanism, Trajectory Preserving and Aggressive Capping, though both operate in a similar manner. A lower runtime bound is computed based on the runtime of the best configuration in addition to some slack. The bound computed by Trajectory Preserving adaptive capping makes use of only the runtime of the best configuration encountered during the current iteration of the search procedure. This approach does not change how the search procedure progresses while still reducing the evaluation overhead. Aggressive Capping imposes a tighter bound by using the solving times from the overall incumbent (globally) to terminate expensive runs. Though this method alters the trajectory of the search, it has been shown to improve the overall configuration procedure by allowing more search iterations.

The combination of these methods allowed ParamILS show very impressive results on a variety of solvers, sometimes improving performance over manual configuration by orders of magnitude. Of particular note is ParamILS' ability to successfully configure the highly parameterised mixed integer programming solver CPLEX [IBM14].

GGA and GGA++ The Gender-Based Genetic Algorithm (GGA) uses a genetic, population based approach to algorithm configuration [AST09a]. As is common with genetic algorithms GGA models each configuration as a genome creates new candidate configurations by means of crossover. A novelty in GGA's approach is that the algorithm uses two gender pools, a competitive and non-competitive pool, with different selection pressures to drive the evolutionary procedure. This approach servers two purposes, to limit the time spent on expensive target algorithm runs, and also to ensure certain level of diversity is maintained. Configurations are assigned to the two pools at random. The configurations in the competitive pool compete for the right to mate (perform crossover) by racing to solve instances in the training set. Once a certain fixed percentage of configurations from this pool has solved the instances, the other runs are

terminated. This approach ensures that the configuration budget is used by the best configurations rather than wasting evaluating poor configurations. The configuration in the non-competitive pool do not compete and are instead included to ensure diversity is maintained. Offspring are produced by selecting a parent configuration from each pool and combining these by means of crossover. The offspring is assigned to either the competitive or non-competitive pool at random. Configurations that have remained for more than a specified number of iterations are removed.

GGA++ extended this idea by using surrogate models to genetically engineer superior offspring during the crossover process [AMS+15]. A surrogate model was built to predict configuration performance using a specially designed random forest. This random forest was constructed to more accurately predict the performance of high performance configurations at the expense of more general accuracy. The surrogate model was then used to select the most promising offspring amongst the many potential crossover possibilities. This paper also proposed using the surrogate model to choose more attractive parents from the non-competitive pool, however this approach was shown to under perform in empirical evaluations.

Sequential Parameter Optimization Sequential Parameter Optimization (SPO) was one of the earliest algorithm configuration procedures to champion the sequential model-based optimization approach [BBLP05]. SPO begins by sampling design points (in the algorithm configuration case configuration vectors) from the design space by means of a Latin hypercube design. The number of design points to sample, d, and the number of response values to evaluate these design points on, r, is specified by the user. The results of these sample runs are used to compute the value of the performance metric we wish to optimise (e.g. mean, quantiles etc.). This provides us with tuples that map the design points to the values for optimisation. A noise free Gaussian process model is then fitted to resulting (design point, response value) tuples. The resulting model is probed by sampling 10,000 design points uniformly at random and the n best according to an expected improvement measure are the response value of these is measured r times. The current incumbent is included amongst the set of configurations to be evaluated. At this point the new incumbent is determined based on the chosen performance metric. If the incumbent is one which has previously been encountered r is increased. Then the process begins again by fitting another noise-free Gaussian process model to the newly evaluated points.

Two important extensions have been proposed for the SPO procedure to improve its performance and general applicability to algorithm configuration. The first, SPO+ [HHLM09], introduces two changes: Firstly it improves the model quality by

performing a log transformation of the response variable. Secondly, it alters the way in which potential incumbents are evaluated to ensure that any new incumbent is the design point with the most response value evaluations. It achieves this using a doubling intensification method inspired by FocusedILS [HHS07].

The second extension, Time-Bounded Sequential Parameter Optimization (TB-SPO) [HHLM10] introduces a number of additional improvements. It removes the Latin hypercube initialisation procedure and instead fits the initial model to a single point (in the case of algorithm configuration, the default configuration). Future points are sampled alternately at random, and using the expected improvement measure based on the trained model. This work also determines the period between model updates based on the cost of updating the model. Further to this the work reduces the cost of training the model by training on a random subset of the previously encountered points, rather than an ever growing training set (a technique called projected process (PP) approximation).

SPO and its extensions demonstrated promising results, however, it was limited to optimising numerical parameters on single problem instances.

SMAC Sequential Model-Based Algorithm Configuration (SMAC) was created with the aim of addressing the limitations of SPO in order to make it generally applicable to the algorithm configuration context [HHL11]. The paper introducing SMAC first defines a simple time-bounded Sequential Model-Based Optimization (SMBO) framework. This framework consists of an initialisation procedure to select the initial incumbent. This followed by three steps in a loop until the configuration budget is exhausted: a model fitting procedure, a configuration selection procedure (based on the fitted model), and an intensification procedure to compare new configurations against the current incumbent and update as necessary. TB-SPO is an instantiation of this framework. The paper proceed by introducing two concrete instantiations of the SMBO framework: Random Online Aggressive Racing (ROAR) and SMAC.

ROAR is a simple model-free instantiation of the SMBO framework with the primary goal of demonstrating how TB-SPO's intensification procedure can be extended to handle multiple problem instances. The intensification scheme used in ROAR relies on the property that variance of a comparison between two configurations is better reduced by evaluating a larger number of instances once than by evaluating a single instances multiple times [Bir05]. The intensification procedure is provided with a list of configurations to compare against the incumbent. Each time a new configuration is compared against the incumbent an additional run on randomly selected (instance,

seed) pair is performed by the incumbent. Following this the candidate configuration iteratively performs runs on the instances previously solved by the incumbent using a doubling scheme. These evaluations proceed until either the candidate configurations performance lags behind that of the incumbent, resulting in elimination, or the candidate solves an equal number of instances to the incumbent with equal or better performance and achieves promotion to incumbent. This intensification scheme is similar to that used in FocusedILS except that the order the instances the candidate configuration must solve is random instead of fixed for each iteration. This is done to reduce the sensitivity of the intensification procedure. The rest of the framework is very simple, the initialisation procedure returns the default configuration, fit model is unused as ROAR is model free, and the selection procedure simply returns a configuration from the configuration space uniformly at random.

The SMAC instantiation of the SMBO framework is an extension of ROAR which primarily focuses on improving the model used to select new configuration. As such the initialisation procedure and intensification procedure remain unchanged from ROAR. The model fitting procedure is changed to allow for categorical as well as numerical parameters by replacing the Gaussian process model seen in other SPO implementations with a random forest model [Bre01]. Random forests have previously been shown to be suitable for this task while also provided estimates of the uncertainty in a given prediction [BM04]. Specifically SMAC trains a random forest to learn a joint model consisting of both configuration and instance features with the goal of predicting the log transformed mean runtime (though other target values are possible). The log transformation is applied as it has previously been shown to improve runtime prediction quality [XHHLB08]. By learning a joint model, SMAC avoids complications around providing the same instances for all configurations as training data.

SMAC uses this model to improve the configuration selection procedure by using the random forests uncertainty estimate to identify areas with large expected improvement values (i.e. areas with low predicted cost and high uncertainty). Specifically SMAC uses multi-start search to identify a small number (ten in the paper) of configurations with locally maximal expected improvement. It supplements these configurations with a large number (10,000 in the paper) of uniformly sampled configurations. These are combined and sorted according to their predicted expected improvement. The resulting list of configurations is interleaved with random configurations to ensure diversity while being processed by the intensification procedure. Using these techniques SMAC has achieved state-of-the-art results and is widely used in many configuration contexts today [BBvB17, THHL13, LBMS17].

Golden Parameter Search Recent work investigating algorithm configuration landscapes suggests that in many cases the parameters exhibit uni-modal response values [PH18]. As such, optimisation procedures used in many algorithm configuration systems are unnecessarily complex. Golden Parameter Search (GPS) exploits this property and proposes a relatively simple approach to semi-independently optimise parameters in parallel [PH20]. Specifically, GPS uses the golden section search algorithm to efficiently reduce the bounds around an optimal numerical parameter value. When a new optimal parameter value is identified it is propogated to all other parameter searches. Due to parameter interactions previously identified can become stale, to combat this the parameters weight decays over time and eventually it must be rerun. GPS also exploits the fact that typically a small number of parameters have a large impact on the solver performance by using a bandit approach to determine which parameter to optimise next. By combining these techniques GPS was able to achieve strong performance, even outperforming state-of-the-art configurators on a number of TSP, SAT, and MIP benchmarks.

2.2.1.2 Configuration with Theoretical Guarantees

The configuration methods outlined in the previous section fall under the broad heading of heuristic techniques; they use heuristics to guide their search for improving configurations towards promising areas of the search space. While such techniques have been shown to perform very well in practice, they lack theoretical guarantees and are susceptible to poor performance in the worst case (for example when presented with an adversarial run of instances). For this reason, a number of recent papers have proposed methods which are highly likely to find approximately optimal configurations with a runtime which dominates existing methods in the worst case.

Structured Procrastination In 2017, structured procrastination (SP) was introduced as a way of finding (ϵ, δ)-optimal configurations [KLBL17]. A configuration, c, is (ϵ, δ)-optimal if:

1. Its expected running time over an instance distribution is within a factor of the optimal configuration, c_{opt}. Formally, $R(c) \leq (1 + \epsilon)R(c_{opt})$, where R denotes the expected running time over an instance distribution.
2. There exists a cutoff time threshold, θ, such that the probability of the configuration timing out is less than δ over the instance distribution.

SP runs in an anytime manner; the longer the algorithm is run for the more accurate its estimate of δ becomes. Upon termination, the algorithm returns this δ value in addition

to the selected configuration.

SP maintains a bounded length first in first out queue for each configuration under consideration. These queues are filled with tuples containing a problem instance and a solving time out. Initially each configuration is processed in a round robin fashion, taking the instance and time out at the head of that configuration's queue and attempting to solve the instance within the time out. If the instance is solved with the allotted time, the solving time is used to compute a lower bound for the configuration solving it; the mean solving time over all solved instances. Otherwise, the time out threshold is doubled and the instance is added to the end of the queue. The configuration with the smallest lower bound on expected runtime is selected and the (instance, time out) tuple at the top of it's queue is processed. Using this greedy approach and time out doubling, SP is able to process instances and establish an accurate estimate of a configurations expected runtime without expending more effort than is required.

Kleinman et al. extended this work to add adaptivity to the SP algorithm [KLBLG19]. Adaptivity allows the configurator to use a smaller number of samples to estimate the performance in cases where there is a low variance, the original paper lacked this property, instead treating every input as if it were the worst case.

Leaps and Bounds *Leaps and Bounds*, proposed by Weisz et al. in 2018, builds on the SP approach to provide a similar yet improved configuration procedure [WGS18]. The paper outlines a simpler algorithm which improves on the runtime bound of the SP method by more accurately estimating the runtime budget required for each configuration. The algorithm is adaptive (requires fewer samples when variance is low), but lacks the anytime property (ϵ and δ must be specified up front). The work also empirically evaluates the performance of "Leaps and Bounds" against "Structured Procrastination", showing the total configuration time required by the former to be less on real world SAT instances.

An extension to *Leaps and Bounds*, *CapsAndRuns* estimates a per configuration runtime cap then uses a Bernstein race on the configurations to determine the best one [WGS19]. The paper also outlines a new theoretical upper bound on the expected runtime which is a significant improvement over existing bounds.

It is worth noting that although *Leaps and Bounds*, *CapsAndRuns*, and *Structured Procrastination* provide strong theoretical guarantees, heuristic methods are likely to greatly exceed their performance in most realistic algorithm configuration scenarios.

2.2.1.3 Online and Dynamic Configuration

Online and dynamic configuration have many different names and definitions in the literature. In this section we refer to methods which perform configuration on a stream of instances as *online algorithm configuration* methods. We use the term *dynamic algorithm configuration* methods for algorithms which update their internal settings on-the-fly during solving rather than between problem instances (these are sometimes referred to as *adaptive* or *reactive* methods elsewhere in the literature)[2].

Online Configuration Instance Specific Algorithm Configuration (ISAC) can be considered a hybrid offline and online configuration approach [KMST10]. At its core ISAC uses the GGA configurator to learn configurations for clusters of instances during an offline training period [AST09b]. These clusters are computed offline using the instance features by the g-means clustering algorithm [HE03]. *G-means* is an extension of *k-means* clustering algorithm that removes the requirement to specify the number of clusters, k. During the production phase new instances which are sufficiently close to a cluster are solved using the learned configuration for that cluster. If the instance is not close enough to any cluster it is solved by a fall-back configuration which has decent performance across the entire set of instances.

ISAC cannot be considered an online configurator as both clustering and training occur offline. However, an extension to ISAC, Evolving Instance-Specific Algorithm Configuration (EISAC) [MMO13], proposes a way of detecting, online, when the instances have drifted enough and uses this to trigger reclustering and reconfiguration of the resulting clusters. EISAC is initialised in the same way as ISAC (g-means clustering of instances according to their features, training a configuration per cluster using GGA). Where EISAC differs is that as new instances are being processed it recomputed the clusters. These recomputed clusters are compared to the original using the Rand measure [Ran71]. If the clustering is deemed to be sufficiently different (according to a user defined threshold) then GGA is used to compute new configurations for the clusters. The frequency that this retraining occurs at can be controlled by adjusting the similarity threshold.

Another method of configuring an algorithm online is the *Continuous Search* paradigm [AHS10, AHS12]. This work uses the downtime between solving instances to find continually improving search heuristics for constraint programming without using an explicit training period or instances. *Continuous Search* begins with no information

[2]Dynamic configurations adapt to the state of the search online, however, the control policy is often learned or provided offline.

about the quality of the available heuristics (as such it uses the default heuristic), it then adopts a lifelong learning approach that learns which heuristic to use and adapts to changes in the incoming instances as it processes the stream of instances.

This is achieved by using two modes: a production (or exploitation) mode, where incoming instance are solved using the current heuristic model, and a learning (or exploration) mode where the system uses available idle time to evaluate alternative heuristics on the last encountered instance. The new information gained from running these evaluations is added to the training set and used to update the heuristic model. Specifically, a support vector machine is trained to predict whether or not the heuristic is better than the current heuristic model (binary classification). The description vector used to train this model consist of both instance and heuristic features. In this way solutions are returned quickly when required while also improving and adapting the heuristic model over time.

Dynamic Algorithm Configuration *Continuous Search* has also been shown to be applicable to the dynamic configuration context in a similar way [AHS10, AHS12]. This technique is applied to a constraint-based solver using search restarts. Restarts occur when the search has backtracked a user specified number of times. At the beginning of the search and every time a restart occurs a description of the instance and search state is computed encompassing both static features (problem definition, variable size and degree information, constraint information etc.) and dynamic features (global statistic on search progress as well as local statistic on the evolution of a given strategy). This description is used to update update the heuristic selection model in the same way as the online variant described above. The major difference is that instead of deciding on the heuristic to use for a particular instance the model now predicts which heuristic to use for the window between search restarts.

Recently a *Dynamic Algorithm Configuration* framework has been proposed which formulates the problem as a contextual Markov decision process (MDP) [BBE+20]. The MDP is *contextual* in the sense that it takes instances into account, specifically creating multiple MDPs with shared state and action spaces but differing transition and reward functions. The states of the MDP are defined by search and instance states at a particular timestep (similar to the description used in *dynamic Continuous Search* for model learning). The action states of the MDP consist of how the configurator can assign parameter values. The transition function is the probability of reaching a certain state in the next timestep by applying a particular action to the state at the current timestep.

This work proposes to use reinforcement learning to optimise the policies based on a reward function. This reward function can be dense i.e. available at every timestep like distance to goal, or it can be sparse e.g. runtime at the end of an algorithm run. Two separate pieces of work have demonstrated how this framework can be employed to dynamically configure well known optimisation algorithms parameters: learning step-size adaptation for the covariance matrix adaptation evolution strategy (CMA-ES) and learning heuristics for the *Fast Downward* planner [SBA$^+$20, SBH$^+$20].

In addition to the two frameworks outlined above, *Continuous Search* and *Dynamic Algorithm Configuration*, a large number of other methods for adapting parameter values during solver or algorithm execution have been proposed. Some well known examples include reactive tabu search [BT94], dynamic restart policies [KHR$^+$02], and greedy randomized adaptive search [Res09] (see [BB07] for a more complete overview.)

2.2.2 Algorithm Selection and Portfolios

Closely related to algorithm configuration is the algorithm selection problem [Ric76]. This technique exploits the complementary nature of many combinatorial solvers and the fact that often there is no single best solver for all instances[3] [XHHL12]. The algorithm selection problem asks, given a set of algorithms and problem instances to identify a performance mapping such that the best algorithm for each instance can be selected. More formally, the problem can be defined as follows [Ric76, KHNT19]:

Given

- a set P' of instances of a problem P,
- a set of algorithms $A = A_1, ..., A_n$ for solving P,
- a performance metric $p : A \times P' \rightarrow \mathbb{R}$ that measures the performance of any algorithm $A_j \in A$ on instance set P',

construct a selector S that maps any problem instance $x \in P'$ to an algorithm $S(x) \in A$ such that the overall performance of S on P' is optimal according to the performance metric p.

Figure 2.9 shows the structure of a typical contemporary algorithm selection model [Kot14]. The selection model S uses features representing the problem space P and machine learning techniques trained on some training data to learn a mapping

[3] Algorithm selection is not strictly limited to combinatorial optimisation (e.g. ensemble methods for machine learning), however, for the purposes of this review we will primarily present it in this context.

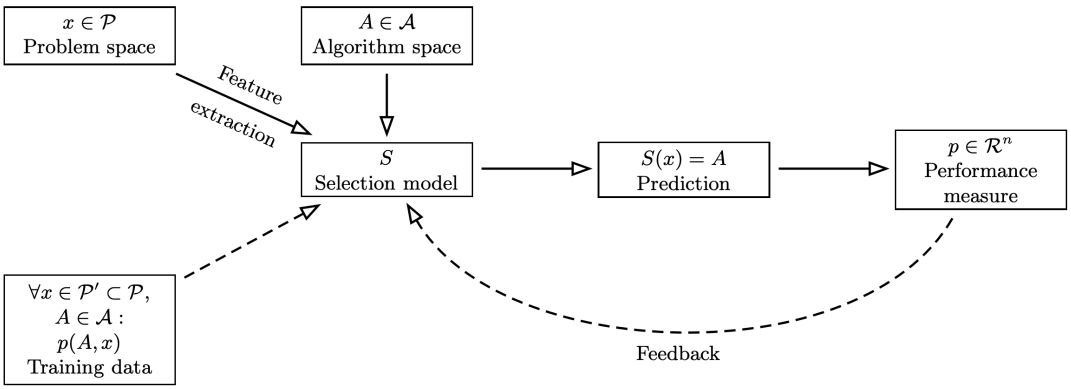


Figure 2.9: A model of the algorithm selection problem [Kot14]. Dashed lines represent optional links.

from a problem instance x to one of the algorithms in A such that performance p is maximised.

By adopting this framework algorithm selection based solvers have enjoyed a great deal of success in many domains [KKHT15, HKMO14c, AGM14], most notably SATZilla which took first place in several tracks at the SAT Competition 2009 and SAT challenge 2012 [XHHLB08, XHS$^+$12]. In fact these techniques became a victim of there own success and were banned from these competitions for a number of years and later relegated to their own track. More recently, in an attempt foster innovation in general portfolio and algorithm selection research rather than finely tuned solutions for particular domain algorithm selection specific competitions have been run in 2015 and 2017 [LvRK19]. Given the abundance of literature written about algorithm selection and related problems we limit this review to approaches which have had the most impact. However, readers who wish to further explore this area are invited to read the more comprehensive algorithm selection surveys by Smith-Miles, Kotthoff, or Kerschke et al. [Smi08, Kot14, KHNT19].

2.2.2.1 Offline Selection

SATzilla The per-instance approach SATzilla is one of the best known algorithm selection systems. SATzilla 2007 gained a large amount of attention by winning multiple medals in the 2007 SAT Competition [XHHLB08]. It uses the training data to determine pre-solvers which attempt to solve the problem instance before feature computation takes place (as this can be an expensive undertaking). A backup solver which has the best average solving time on the training set is also identified. Training instances which the pre-solvers fail to solve are used to train algorithm specific hierarchical hardness models.

Separate ridge regression models are trained for SAT and UNSAT instances. Then a Sparse Multinomial Logistic Regression (SMLR) classifier is trained to predict the probability that an instance an instance is SAT or UNSAT. Finally, the ridge regression model's predictions are combined, weighted according to the probability output by the SMLR classifier. A subset of solvers which achieve the lowest total runtime is selected.

During the online phase the pre-solvers are used initially to attempt to solve the instance. Failing that, features are computed and used by the hierarchical hardness model to identify the fastest predicted solver to solve the instance. If for some reason features cannot be computed the backup solver is used for solving.

Satzilla 2012 replaces the hierarchical hardness model with a cost-sensitive sensitive pairwise classification approach [XHS+12]. This approach trains a classifier for each pair of solvers to predict which will provide the best performance. These classifiers take into account the impact an incorrect classification has on the performance. The solver with the most votes across all pairwise classifiers is selected to solve the instance.

3S *3S* proposes a semi-static solver schedules approach [KMS+11]. This approach combines a nearest neighbour based algorithm selection approach with a static schedule of solvers. The rational behind this is to hedge against incorrect selections made by the algorithm selection procedure by also running a static schedule of solvers for a percentage of the solving budget (in the paper 10%). The algorithm selector partitions the training set using *g-means* clustering (similar to *ISAC*) [HE03]. For each cluster the best k value is identified based on the weighted distance. The solver with the lowest solving time on the k nearest neighbours is selected. The static schedule uses column generation to calculate a set of solvers and solving budgets which will solve the most training instances. One notable aspect of this approach is that it achieved excellent performance (seven medals including two gold) across tracks in the 2011 SAT competition without tailoring its approach to each specific track (i.e. it trained once on a fixed set of instances and solvers). This demonstrated the strength of a low-bias/non model-based approach.

Cost Sensitive Hierarchical Clustering (CSHC) CSHC builds on 3S by replacing the k-nearest neighbours clustering approach with a cost sensitive hierarchical clustering method [MSSS13a]. CSHC retains the static schedule of solver run for 10% of the solving budget. The proposed clustering method starts with all methods in a single cluster and recursively identifies hyperplanes until the cluster size reaches a certain size threshold (in this work, 10). Each misclassification in a cluster incurs a cost, the hyperplane is chosen to reduce the aggregated cost per cluster.

More recent work has extended this approach by introducing the idea of recourse to the selection procedure [AST18]. *Recourse-CSHC* determines the confidence in the selectors choice by comparing the performance of the chosen solver to the average performance of all solvers on the neighbouring instances in terms of standard deviations. If the selected solver's performance greatly exceeds that of the others the selector can be confident in its choice. If not the selector opts to use either a predetermined static schedule of solvers that exhibit strong performance over all instances for a period of time or a dynamic schedule which selects solvers and their budget based on the problem instances nearest neighbours. *Recourse-CSHC* also exposes many of the parameters controlling the selection process and tunes these using the *GGA* configurator.

ASAP V2 and V3 ASAP v2 and v3 ranked first and second in the Algorithm Selection competition 2017 respectively. Similar to *3S* and CSHC, ASAP combines a global presolving schedule with an instance specific selector [GSS17]. Unlike the aforementioned approaches, ASAP identifies an optimal split between the computational budget for the pre-scheduler and algorithm selector by formulating the problem as a bi-objective optimisation problem and identifying the knee point using heuristics. Due to the interdependence of the pre-scheduler and algorithm selector (the selector should excel where the pre-scheduler fails and vice versa), ASAP incrementally trains both in alternate phases as follows:

1. Build a scheduler which maximises the number of training instances solved in a short time budget.
2. Train a random forest regression model on a per algorithm basis to predict solver performance on the training instances. The outcomes from Step 1. are used as additional features.
3. Create the final pre-scheduler to optimise the number of instances solved with less weight given to instances which can be solved quickly using the selector from the previous step.
4. Train a performance model over all training instances again using the results off the pre-scheduler construction as features. This model will then serve as the algorithm selector.

The primary difference between ASAP v2 and v3 is the number of solvers in the pre-scheduler is fixed to three in version 2 and optimised to a value between one and four in version 3.

CPHydra CPHydra adopts a case-based reasoning approach to portfolios and achieved first place in the CSP solver competition in 2008 [OHH+08a]. During training CPHydra solves a set of training instances using the solvers in the portfolio to construct a case base. In the online phase the most similar for each of the incoming instances are identified (using a weighted k-nearest neighbours, k=10). A constraint program is used to maximise the number of cases solved in the given time budget.

Proteus Proteus exploits the ability to encode CSP problems as SAT in order to extend the pool of high quality solvers across problem domains [HKMO14c]. The selector uses a hierarchical model to select which encoding and solver to use. At the root it decides whether the instance should be solved in it unaltered CSP form or converted to SAT. If the problem is converted to SAT the selector then decides which SAT encoding to use (direct, support, order). Finally, when the problem representation is decided, the selector chooses the optimal solver to solve the instance. The authors conduct extensive studies at each branching point to decide the which model and set of features to use for each decision. The best performing approach uses regression models (M5P and Linear Regression) and mostly CSP features (SAT direct-order features are used in a single case).

2.2.2.2 Online Selection

SUNNY *SUNNY* builds a schedule of CSP solvers in a portfolio online without any prior training [AGM14]. *SUNNY* finds similar instances to the current instance using k-nearest neighbours. It then selects a minimum subset of solvers that could the solve the greatest number of neighbouring instances and schedules them based the number of neighbouring instances solved.

SUNNY-AS2 is an improved version of *SUNNY* with a number of enhancments [LAMG20]. It capable of handling multiple domains and appeared in the Algorithm Selection competition 2017 finishing in 3rd behind the ASAP algorithm selectors. The two main enhancements in *SUNNY-AS2* are a wrapper based feature selection method to remove unhelpful instance features, and a method for tuning the value of k which determines the number of neighbours to choose.

Multi-Armed Bandit Approaches A common approach to online algorithm selection is to treat the problem as a multi-armed bandit problem. Here, the goal is to estimate the underlying reward distribution given a fixed number of evaluations while attempting to maximise the reward achieved. In the context of the online algorithm

selection problem, each incoming instance provides an evaluation opportunity while solving the instance (potentially considering solving time) is the reward.

Gagliolo presents an online selection system, *GambleTA*, which uses a multi-armed bandit approach to select between time allocators (parallel portfolio schedulers) [GS06]. This is modelled as a multi-level problem where the lower level constructs time allocators based on the observed reward (solving time) and the higher level is tasked with selecting between these time allocators. EXP4 is used as the bandit problem solver [ACFS02]. The downside of this approach is that EXP4 cannot handle unbounded losses, as such it is necessary to arbitrary bound on algorithm runtimes which invalidated the optimal regret property of the solver. An extension to this work proposes a simpler version of the bandit solver framework which is able to handle unbounded losses [GS11].

DeGroote et al. outline another approach to online algorithm selection using contextual multi-armed bandits to select which solver to run [DCBK18]. Initially a set of random forest regression models are trained per algorithm using instance features to estimate performance on some offline training data. During the online phase a multi-armed bandit approach using the trained models selects which algorithm to run. This work evaluates three selection schemes: greedy which always chooses algorithm with the best predicted performance, and two common approaches from the literature which aim to balance exploration and exploitation, ϵ-greedy and upper confidence bound (UCB) [BC12]. Surprisingly, they find that the greedy approach outperforms the balanced approaches. The models are updated periodically using only the information gained from the selected algorithms. The work shows that even a relatively simple online learning approach is competitive with the state-of-the-art offline systems on many scenarios in ASlib [BKK$^+$16].

2.2.3 Combined Algorithm Selection and Configuration

Given the close relationship between algorithm selection and algorithm configuration it is unsurprising that a number of combined approaches have emerged.

Hydra Hydra builds a portfolio of solvers configured so that their performance is complementary [XHL10]. It achieves this by adapting the performance metric used when configuring the solver. The solvers performance is taken if it exceeds that of the portfolio otherwise performance performance of the portfolio is used. This drives the configurator to find configurations which improve the overall quality of the portfolio without penalising configurations which are outperformed by another solver and would

not have been selected anyway.

ISAC++ In this work *ISAC* is improved by altering adopting an algorithm selection instead of k-nearest neighbours to determine which configuration to use [AGMS16]. Specifically, *ISAC++* clusters instances and configures a solver per cluster using GGA as in the original algorithm. Where *ISAC++* differs is that it uses cost-sensitive hierarchical clustering to decide which configured solver to use. Using this scheme *ISAC++* is able to outperform *Hydra* and win multiple categories in the 2013 and 2014 MaxSAT Evaluations.

Cedalion *Cedalion* uses algorithm configuration to build a sequential planning portfolio [SSHH15]. It does this in a greedy iterative fashion. Starting with an empty portfolio, a configurator (SMAC in this work) is used to configure a solver on a set of planning instances. The solving time budget is added to the configuration space allowing the configurator jointly optimise both the configuration and solving time. The learned (configuration, time) pair are added to the portfolio and all instances solved by this pair are removed from the training set. At this point a new configuration is learned and the procedure continues until convergence or the solving budget has been exhasuted.

AutoFolio *AutoFolio* uses algorithm configuration to automatically optimise both the choice of algorithm and algorithm parameters for a highly parametric solver [LHHS17]. Specifically they use the algorithm configurator SMAC to configure Claspfolio 2 algorithm selection framework on a variety of scenarios from ASlib [HHL11, HLS14, BKK$^+$16]. Remarkably, this simple approach of applying an existing configurator to an existing highly parametrised algorithm selection framework yields significant improvement in ten out of thirteen scenarios evaluated and state-of-the-art performance on seven of those ten.

CASH Combined algorithm selection and hyperparameter optimisation (CASH) employs a similar idea applied to the field of automated machine learning. AutoWEKA exposed the machine learning library WEKA's models and hyperparameters as configurable parameters to produce a 786-dimension parameter space [THHL13, KTH$^+$17]. This is then optimised using SMAC or a Tree of Parzen Estimators approach to maximise performance based on some chosen metric (e.g. mean squared error) on a set of examples [HHL11, BBBK11]. AutoSklearn uses the same idea to fit models from the well known scikit-learn machine learning library [FKE$^+$15, FEF$^+$20, PVG$^+$11].

confStream confStream is a combined algorithm selection and configuration system for stream clustering algorithms [CTBP20]. It uses a sliding window approach to determine which algorithm to use for the next window. Candidate algorithms and configurations are evaluated on a set of test instances from the previous window to determine their merit. Configuration occurs by taking an algorithm from the portfolio as a parent this parent's parameter values are altered by sampling from a biased probability distribution similar to F-Race in order to produce a child configuration [BYBS10].

2.2.4 Runtime Prediction, Parameter Importance, and Learning

2.2.4.1 Runtime Prediction and Analysis

Previously we have seen how empirical performance models (EPMs) which predict a solvers runtime based on solver, instance or configuration features form the core of many algorithm selection and configuration systems (see e.g. [XHHLB08, HHL11]). In this section we will briefly review the literature related to runtime prediction and its applications.

Leyton-Brown et al. outline the methodology for training EPMs and run a case study on using EPMs to predict the runtime of combinatorial auction instances in [LNS09]. Hutter et al. show that random forests and Gaussian processes provide strong predictive performance across a range of combinatorial problems [HXHLB14]. This work also introduced a new set of descriptive features for SAT, MIP and TSP instances.

Hurley et al. argue that due to the stochastic elements present in many solvers a single run is insufficient to characterise the performance of a solver on an instance [HO15]. This work demonstrates that in the SAT competition 2014 there is significant overlap between the runtime distributions of the top three solvers, and that any ordering could feasibly have occurred. Statistical bounds are presented for each of the solvers. Adopting statistical over single point predictions is offered as a potential antidote for the problem in the case of runtime prediction while comparing on the basis of runtime distributions is suggest for empirical evaluations.

Arbelaez et al. show that it is possible to estimate the runtime performance of parallel local search algorithms by analyzing there sequential runtime distribution [ATC13, TARC16]. Similarly, [ATO16] presents a way of predicting the runtime distribution of sequential and parallel local search solvers on an instance. This is achieved by identifying a suitable distribution, fitting the distribution's parameters using a regression model, and applying order statistics to predict the runtime distribution of the parallel algorithm.

Surrogate models, which use EPMs to predict the runtime of a solver solving an instance, have been proposed as a way of greatly reducing the time needed to evaluate configuration algorithms [EHHL15, ELH+18]. This work outlines how best to collect training data and use it to learn a surrogate model which can be use as a replacement for expensive solver evaluations. Practical considerations such as how to handle right-censored data (due to runtime caps), and model performance are also discussed.

2.2.4.2 Instance Ordering and Improved Learning

Two concepts related to instance ordering outside algorithm selection and configuration domain which have received significant attention are *Curriculum Learning* and *Self-Paced Learning* [BLCW09, KPK10]. *Curriculum Learning* advocates ordering problem instances from simple to complex to increase the learning rate of the model (neural networks in this work). The work demonstrates using toy examples the effectiveness of this approach, both in terms of model accuracy and convergence rate. Closely related to this work is the concept of *self-paced learning*. While *curriculum learning* defines the curriculum (instance ordering) prior to solving, *self-paced learning* proposes a dynamic approach where feedback about the model's current ability is used to determine which instance (based on difficulty) is supplied next. Jiang et al. note that neither system is perfect; *curriculum learning* fails to account for feedback from the learner while *self-paced learning* may be prone to overfitting as it is entirely driven by training loss [JMZ+15]. They propose a unified framework, *self-paced curriculum learning*, combining both approaches and balancing self-pacing using a regularisation term.

Returning to the algorithm configuration domain, a similar concept is proposed by Styles et al. to improve configuration scaling performance [SHM12]. Because algorithm configurators typically require a large number of evaluations to achieve sound performance, difficult instances in the training set can stymie the configuration procedure. The approach partitions the train instances into easy and intermediate difficulties (reserving hard instances for testing). The work shows that configuring on the easy set of instances then using intermediate instances to validate and select which configurations to use for solving the difficult instance outperforms training and selecting based on a single class of instances only.

Order racing protocols extend this work by using racing protocols for the validation and selection procedure [SH13]. Configuration takes place as normal using the easy instances. For validation, the intermediate instances used for selection are ordered by difficulty (based on default configuration solving time) and solved using a racing procedure which removes configurations when there is enough statistical evidence to

do so. This greatly reduces the cost of validation by gathering evidence using the intermediate instances which can be solved fastest and using this to remove under performing configurations. The work evaluates two racing methods, a variant of F-Race, and a novel approach based on permutation tests showing the latter to be more effective.

Another approach proposed to improve the performance of algorithm configurators is warmstarting [LH18]. This work proposes using previous configuration runs to warm-start the current configuration procedure in two ways: by initialising the configurator with a set of strong configurations based on previous configuration runs and using the data from previous runs to train the initial model (this only applies to model-based approaches). The initialisation procedure presented adds configurations to the pool in a greedy manner by selecting the configuraions which reduce the cost of the pool the most, this is similar to the method Hydra uses to construct portfolios [XHL10]. To warmstart the model individual EPMs are trained on data from previous configuration runs and combined with a model trained on the current data using a linear model optimised to reduce the combined models root mean squared error.

2.2.4.3 Parameter Importance

Not all parameters have the same impact on solver performance. In this section we outline a number of methods which have been created to identify critical parameters which influence the solver the most.

Forward Selection In the first work that we are aware of which aims to assess the importance of parameters in the configuration, forward selection is used to identify a subset of parameters which yield strong predictive power of the solvers performance [HHL13]. This approach trains regression models using an increasing set parameters (incrementally adding one parameter at a time) using data collected by evaluating randomly selected configuration and instance pairs. The downside of the forward selection approach is that it is very expensive, requiring a model to be trained for each free parameter. This work shows that relatively few parameters (as little as two) account for the bulk of the solver performance and that relatively simple models can accurately predict this performance. A method for assessing the relative impact of the selected parameters is also proposed by observing the reduction in prediction performance by removing the parameter from the set used for prediction.

fANOVA A more efficient approach to guaging a parameters impact using functional ANOVA is outlined in [HHL14]. Here a random forest regression model is trained using runs of the configurations to predict performance. This model display the relationship

between configurations and performance. This model is decomposed using functional ANOVA to identify parameter importance and the interaction between parameters.

Ablation Analysis Brute force ablation analysis is another method used to identify parameter importance [FH16]. This method analysis the path between two configurations for example the default and the configurator incumbent by iteratively modifying one parameter at a time and retaining the configuration with the best performance. This allows the procedure to identify changes which result in improvement and those which are a by-product of the configuration process. Modifying a single parameter at a time incurs a major computational overhead (over 100 days CPU time are reported in the paper). To address this, the authors propose a racing approach based on F-Race [BYBS10] to reduce the number of configurations required. An extension to this which uses surrogate models in place of full evaluations renders the approach feasible for typical use cases [BLE+17].

Tools In addition to the methods outlined a number of tools have been created to help with the analysis of the configuration space. SpyBUG is a tool designed to help identify invalid or erroneous configurations which can cause solvers to fail [ML16]. SpySMAC and its successor CAVE are tools which make the analysis of parameters easier by generating plots, visualisations and reports based on configuration runs [FLH15, BMLH18].

2.3 Chapter Summary

In this chapter we provided an extensive overview of the relevant literature required to place this thesis in context. Specifically, we outlined the techniques, methods and heuristics used to solve various combinatorial optimisation problems such as the *Constraint Satisfaction Problem*, *Boolean Satisfiability*, and *Integer Programming*. The chapter also presented an overview of techniques for automatically selecting, combining, and optimising the options available when solving hard combinatorial problems so that performance is maximised.

Chapter 3

Real-time Configuration Framework

Summary. *This chapter motivates the need for real-time algorithm configuration. We show how, with the advent of modern multi-core systems, tuning solver parameters while solving problem instances with no increase in wallclock time has become an achievable goal. A high level overview of our proposed real-time algorithm configuration framework is presented, as well as an outline, in broad terms, of the functions of each of the constituent components of this framework.*

The work presented here has appeared in the peer-reviewed conferences AAAI 2014 [FOMT14], SoCS 2014 [FMOT14] and IJCAI 2015 [FMO15].

3.1 Motivation

3.1.1 Prevailing Configuration Methodology

In Section 2.2.1 we saw how automatic black-box algorithm configuration is able to remove the burden of manually configuring algorithms with tens or even hundreds of parameters from the end-user. This frees up time to spend on more important work while eliminating the need for extensive domain knowledge of the algorithm's parameters and potential interactions. Automatic algorithm configuration allows the designers of algorithms to extract the best possible performance from their algorithms by tailoring the algorithm settings to the particular context where it will be applied. This in turn enables a fairer comparison between algorithms.

These advantages are achieved by using computers to systematically evaluate the configuration space and shift the search towards areas where improvement is more likely to

be discovered. A number of different methods for this have been proposed including racing [BYBS10], genetic algorithms [AST09b], iterated local search [HHLBS09], and sequential model based optimisation [HHL11], to name but a few. While these configurators function very differently, they all use the same offline configuration methodology. Here a set of representative training instances is collected, the configurator is assigned a time budget during which it trains on these, and then produces a configuration that is used to solve all new instances going forward.

3.1.2 Exploiting Parallelism for Real-time Configuration

In many practical applications of combinatorial optimisation it may not be possible to fulfil the requirements of the offline configuration methodology. This could be either because a representative set of training instances is unavailable or there is not enough time to perform the necessary training.

In many common business scenarios, such as delivery routing or advertising space auctions, instances will arrive in the form of a stream. The business' ultimate goal is to find a solution in the minimum amount of time possible. Over time the instance distribution of this stream may shift, for example due to an uptick in deliveries caused by a global pandemic or increased competition for advertising space at Christmas. This can cause an unwelcome degradation in solving performance. The obvious solution is to (re)configure the solver for this these new instance distributions. However, doing so requires that the business collect a sufficient sample of these new instances to train on and allocate enough time to learn the new configuration (during this time solving performance will still be impacted).

Here we propose a variant of the algorithm configuration problem that we call real-time algorithm configuration aimed at providing a solution for the use case outlined above. Real-time algorithm configuration combines the process of finding a solution and configuring the solver while processing a stream of instances. It is a solutions first approach to algorithm configuration; solutions should be returned to the user in the minimum amount of (wallclock) time possible (or within a user defined slack period of this minimum). Improving the solver configuration is a secondary, but nonetheless important, goal of real-time algorithm configuration.

In this thesis we show that due to the prevalence of modern multi-core systems it is now possible to build such a system. Moore's Law says that the transistor count in an integrated circuit will double every two years [M$^+$75]. This law has held true to this day. However, due to heat issues that arise from increasing single core clock

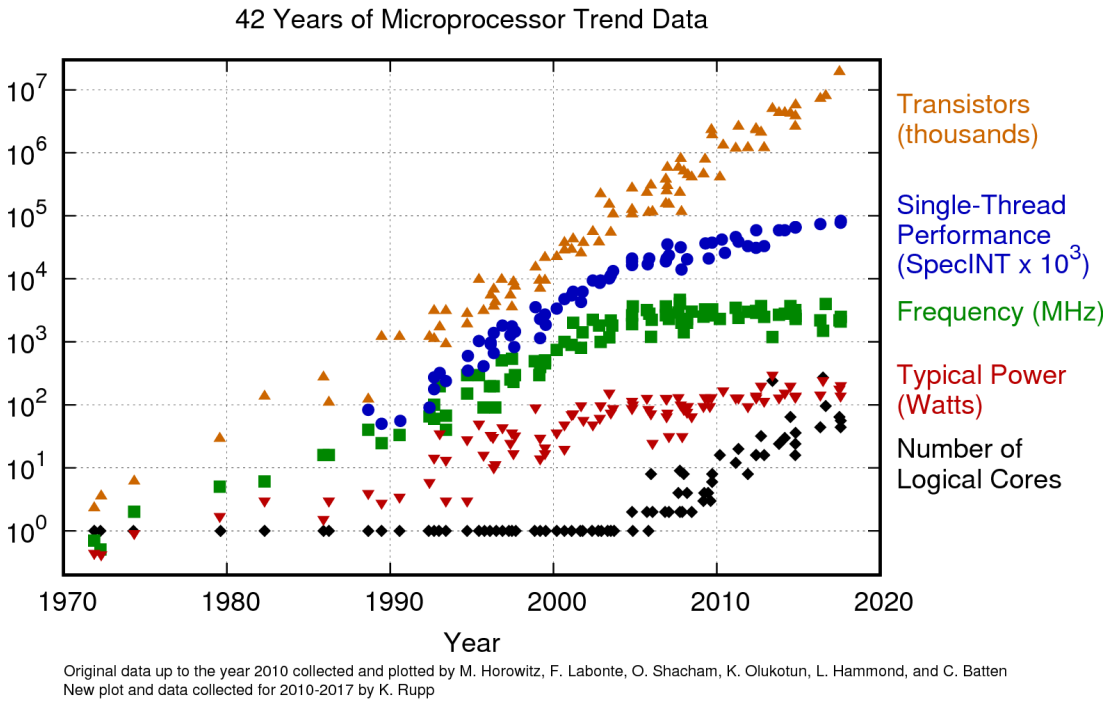


Figure 3.1: CPU trends from 1975 to 2017. Credit [Rup18].

speeds, increases in transistor count have been accomplished through additional CPU cores rather than higher CPU frequencies in recent years. Figure 3.1 provides a strong illustration of this trend [Rup18]. We see that the number of transistors (orange) continues to rise exponentially while processor clock speed (green) and performance (blue) have stagnated or show only a modest increase since 2005. As a counterbalance to this the number of logical cores (black) have begun to increase exponentially since 2005. Today, even consumer CPUs can have a large number of physical CPU cores, for example AMD's latest processor the Ryzen Threadripper 3990X has 64 physical CPU cores. This trend looks set to continue for the foreseeable future and as such algorithms that exploit multiple cores effectively will benefit greatly from the increase in parallelism.

The Real-time Algorithm Configuration through Tournaments (ReACT) framework that we outline in the rest of this thesis exploits the parallelism offered by multi-core systems in order to perform comparisons between configurations with no increase in wallclock time. As additional processing cores in desktops and workstations are often left idle and unused we are allowed increased performance essentially for free in many cases. In cases where there is a cost associated with using additional cores, such as when using a cloud computing service, the ReACT framework still provides value as the reduced overhead due to aggressive run termination means that only the resources required to reach a solution quickly are used. Finally, the framework is still a black-box

configuration method; parallel architectures are exploited at the framework level, rather than the solver level, so no additional instrumentation or modifications are required to the solver other than a lightweight wrapper and configuration space specification.

Before describing the framework and its components we first offer some discussion around the challenges a real-time algorithm configuration system must overcome and the way these shape our decisions around the design of the framework.

In order to compare two or more configurations without increasing the wallclock time we must be able to run configurations in parallel. Therefore it is necessary that the ReACT framework run on a system which is able run at least two solvers. This requirement is increasingly easy to satisfy but still worth consideration. It is also worth noting that while all of the experiments in this thesis are run on a single CPU system, there is no reason that these methods cannot be distributed across multiple machines provided sufficiently accurate communication and measurements can be achieved. Throughout this thesis we also assume instances arrive in a constant stream without break or interruption. Section 3.2.2 outlines how, in this scenario, the parallel racing approach at the core of the ReACT framework is effective at inferring information about the quality of configurations without increasing wallclock time required to receive a solution.

As the primary goal of real-time configuration is to solve the instance at hand as quickly as possible, the methodology and methods used must be as computationally inexpensive as possible. This is particularly important to remember when considering components that typically incur a large overhead such as model training or feature computation.

The requirement to return a solution in the minimum achievable time also limits our ability to compare configurations on the same instance. As instances arrive in a constant stream and solutions must be returned as quickly as possible, we solve every instance once and only once (terminating all other configurations upon finding a solution). As such it is impossible to directly compare different configurations on the same instance. Direct comparison is the method commonly used by offline algorithm configuration to establish a configuration runtime distribution. The ReACT framework instead opts to use a ranking-based approach where only the (typically censored) ranking of configurations is considered. Specifically, the framework uses standard competition ranking where configurations which compare equally (for example in the case of timeouts) receive the same rank number. A consequence of this is that the time taken to solve is not considered; given that each instance is solved only once by a single configuration we cannot determine if the solving time was the result of the instance difficulty, configuration quality, or a combination of both factors. Despite these limitation we show in the rest of this thesis that ranking provides a good enough proxy for traditional runtime

comparison to achieve strong configuration performance.

With these considerations in mind, we dedicate the rest of this chapter to outlining our proposed framework and its constituent components. Section 3.2.1 presents a high-level overview of the framework and how its various parts interact. In Section 3.2.2 we discuss the intricacies of the ReACT framework's parallel racing approach, while Section 3.2.3 describes the framework's configuration pool and leaderboard. Section 3.2.4 shines light on the function candidate selection procedure and, finally, Section 3.2.5 summarises the important aspects of configuration generation and removal.

3.2 ReACT: Framework and Components

The ReACT framework has been designed to be modular and extensible. This allows for future algorithm configuration practitioners and end users to build extensions and improvements easily. In this section we deliver a high-level overview of the constituent components and how they interact with one another. We also attempt to provide some intuition as to why such approaches are effective in practice. However, we leave experiments with concrete instantiations of the ReACT framework to Chapter 4.

3.2.1 Framework Overview

The ReACT framework can broadly be divided into three main parts: a *racing and selection* component, a *ranking system* and *configuration pool*, and *pool maintenance* methods. Within each of these components there are a number of constituent methods. Figure 3.2 illustrates the various parts of the ReACT framework, the methods that make up these parts and the interactions between these methods.

The *racing component* consists of a *runner* that runs the combinatorial solvers with different configurations in parallel. This piece of the system is also tasked with terminating all runs when one of the solvers has found a solution or when the allotted solving time is exceeded. The *runner* also handles parsing the solution and result information so that it can be used by the *ranking system* easily and written to output. The other part of the *racing component* is the *configuration selector*. This uses *leaderboard* information in order to provide a set of configurations to the *runner* (the exact details of this is covered in Chapter 5). The *selection method* uses the available ranking information from the *leaderboard*.

The *ranking component* holds a pool of configurations as well as information about how these configurations are performing relative to one another. This *pool* contains

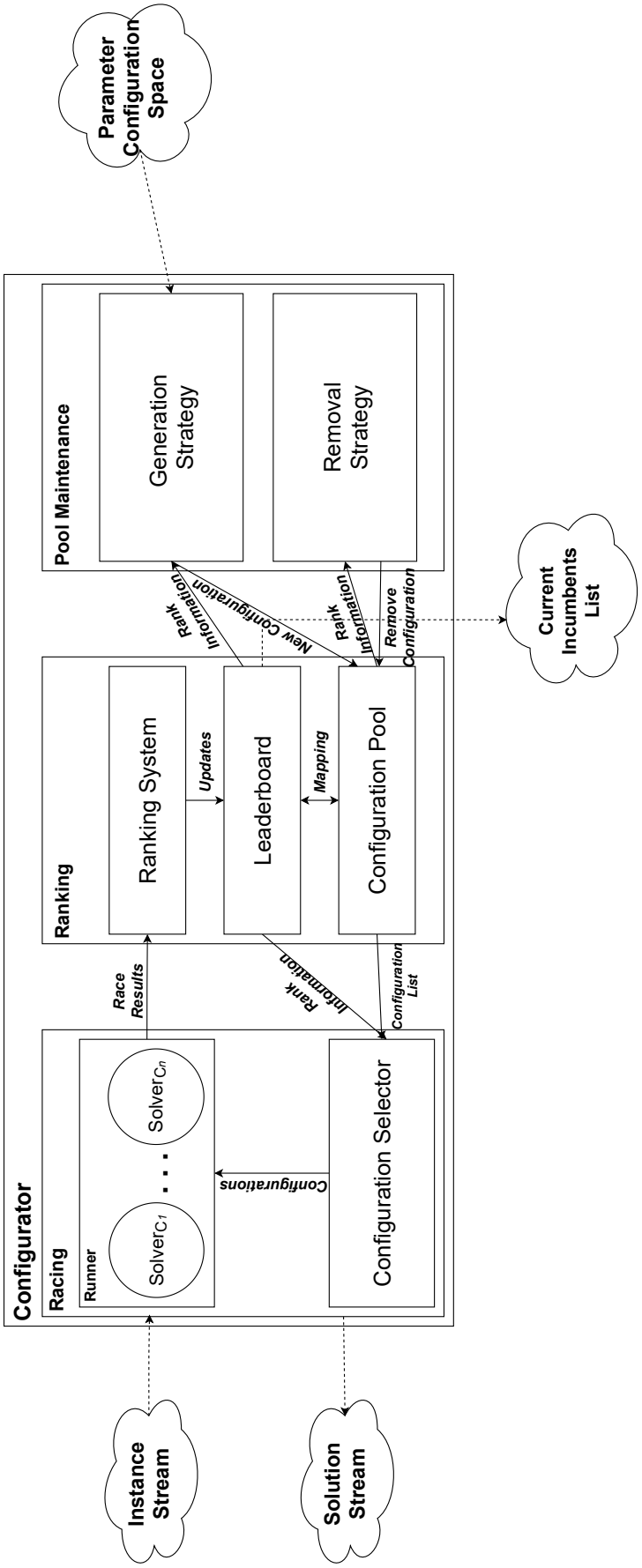


Figure 3.2: An overview of the components and interactions between them that make up the ReACT system.

all configurations currently being evaluated. There is a direct mapping between the *leaderboard* (if ranking is used) and the configurations available in the *configuration pool*. The *leaderboard* is updated by means of the *ranking system* which receives race outcome information directly from the *runner* component. The *leaderboard* is integral to driving the decisions that occur in a number of other methods in the configuration system.

The *pool maintenance* component contains methods for the intelligent addition and removal of configurations to and from the configuration pool. The *removal strategy* evaluates the current state of the *leaderboard* and uses this information to decide which configurations are non-competitive and can safely be removed without impacting the configuration procedure. The *generation strategy* is called to replace configurations that are removed from the *configuration pool* with new configurations that have the potential to perform better. New configurations are generated using parameter values in the domains defined in the parameter configuration space file. This file also specifies dependencies between parameters. There are a number of strategies for generating competitive new configurations. These must carefully balance exploring the space of potential configurations with exploiting information gleaned from existing known good configurations. Strategies for both generation and removal are discussed in more detail in Chapter 6.

Algorithm 7 codifies Figure 3.2 and shows the role of the black box functions on this configuration procedure more clearly. The algorithm's inputs consist of a parameterised solver, A, a stream of instances to solve, Π, the configuration space for solver A, Θ, a set of configuration to include in initial evaluations Θ_{ws}, the maximum time allowed to solve an instance, t, the size of the configuration pool/leaderboard, L_n, and the number of configuration to evaluate in parallel, R_n. The ReACT algorithm processes a potentially infinite stream of instances, as such it does not return a value, however solutions to solved instances and logs detailing the current incumbent configurations are written to file. Function names written in all capitals are open to the user to implement. In Sections 4.2 and 4.3 we outline and benchmark two potential concrete instantiations of this framework.

Lines 11 and 12 show how the configuration pool, CP, is initialised by combining the configurations supplied for warm starting with a set of configurations uniformly sampled from the configuration space in order to produce a configuration pool of size L_n. This configuration pool is used to produce a leaderboard, L, that maps between configurations and the chosen rating system.

With both leaderboard and configuration pool initialised we are ready to begin pro-

Algorithm 7 ReACT Framework

1: **procedure** REACT(A, Π, Θ, Θ_{ws}, t, L_n, R_n)
2: **Input:** A, A parameterised combinatorial solver.
3: Π, A stream on combinatorial problem instances.
4: Θ, The parameter configuration space of the solver algorithm A.
5: Θ_{ws}, A set of configurations from the configuration space, Θ, used to warm start the configuration (these are optional).
6: t, The cutoff time or maximum allowed time for solving each instance.
7: L_n, The number of configurations held in the leaderboard.
8: R_n, The number of configurations to be run in parallel.

9: **Side Effects:** Instance solution, s, written to output file.
10: Evolving list of current incumbents, Θ_{inc}, written to file upon instance solution.

11: $\Theta_{rand} \leftarrow SampleRandomConfigurations(L_n - |\Theta_{ws}|, \Theta)$
12: $CP = \Theta_{ws} \cup \Theta_{rand}$ ▷ Initialise the configuration pool with warm start and random configurations.
13: $L \leftarrow$ INITIALISE LEADERBOARD(CP)
14: **for** instance, π in Π **do**
15: $C \leftarrow$ SELECT CANDIDATES(CP, L, R_n)
16: $R, s \leftarrow RaceParallel(\pi, A, C, t)$ ▷ R is a vector which holds the runtime and solution status of each solver. s is the instance solution, this is output to file.
17: $L \leftarrow$ UPDATE LEADERBOARD(R, C)
18: $CP = CP-$ REMOVE CONFIGURATIONS(L)
19: $CP = CP\cup$ GENERATE CONFIGURATIONS(L, CP, Θ, L_n)
20: **end for**
21: **end procedure**

cessing the stream of instances (Line 14). For each available core the SELECT CANDIDATES function chooses a configuration to run from the configuration pool using information from the chosen rating system (Line 15).

Line 16 shows the function RaceParallel which handles running multiple solvers in parallel with the selected configurations. The first configuration to return a solution sends a terminate signal to the other runs. To account for any discrepancies in start time, and thus ensuring that each approach is evaluated fairly, the termination signal also encodes the time required by the quickest configuration to finish. When all runs have finished or been terminated, the winner is the configuration with the lowest time taken to solve the instance. In the case of optimisation problems an instance is considered solved when a solution is found within a certain, user-specified, tolerance of the global optimum; this is done because proving optimality can take a large amount of time relative to finding the solution. We note that unlike in the GGA configurator [AST09b],

where a given percentage of the participants in a tournament are considered the winners, here we only consider the best configuration as the winner.

The tournaments in ReACT are inspired from the tournaments in GGA. The tournaments serve a particularly important function in ReACT, in that they ensure that the user's solving experience is not affected by the search for more effective configurations. Solutions are provided while configurations are simultaneously evaluated, due to the parallel nature of the approach. As solvers using poorly performing configurations are terminated once a solution is found by any parameter setting in the pool, the user only has to wait as long as it takes for the best parameter setting to find an answer. The solution is output to a file or standard output.

The *RaceParallel* function also handles parsing the solver output to be passed to the ranking system so that the leaderboard can be updated with the latest results. The procedure used to update the leaderboard, UPDATE LEADERBOARD, is dependent on how the leaderboard itself is structured and can be implemented in different ways (Line 17).

In order for the configuration procedure to progress and to prevent stagnation in the configuration pool it is necessary to introduce new and improving configurations where possible. REMOVE CONFIGURATIONS uses the leaderboard to safely remove underperforming configurations thus freeing up space for potentially better candidates. The function GENERATE CONFIGURATIONS often drives improvement by utilising prior information about the quality of the configuration from the leaderboard when creating new candidate configurations. This function is also responsible for deciding how the balance between exploring the configuration space and exploiting known good configurations is handled. Lines 18 and 19 show how REMOVE CONFIGURATIONS and GENERATE CONFIGURATIONS work in tandem to clear and replenish the configuration pool.

In all, this framework provides a strong backbone to create a powerful and extensible online configuration system. The subsequent sections delve each component in more detail.

3.2.2 Parallel Racing

At the core of the ReACT system is parallel racing. This involves racing solvers instantiated with different configurations against each other on the same combinatorial problem instance using the multiple cores available. Such racing allows us to infer a ranking over the configurations without increasing wall-clock time required. This is

possible by terminating runs as soon as possible (after the problem instance has been solved once).

Example 3.2.1. To give a concrete example lets take a synthetic toy problem where we simplify the act of solving problem instances to its essence. We remove instance and solver variation so that each individual configuration has the same single deterministic solving time on every instance. This solving time is sampled from a normal distribution (mean = 50, standard deviation = 12.5) and truncated to between 0 and 100 seconds. Whether solving times follow this distribution is not a concern as the only goal of this example is to evaluate parallel racing (and how this scales) in comparison to other solving schemes in isolation from external influences.

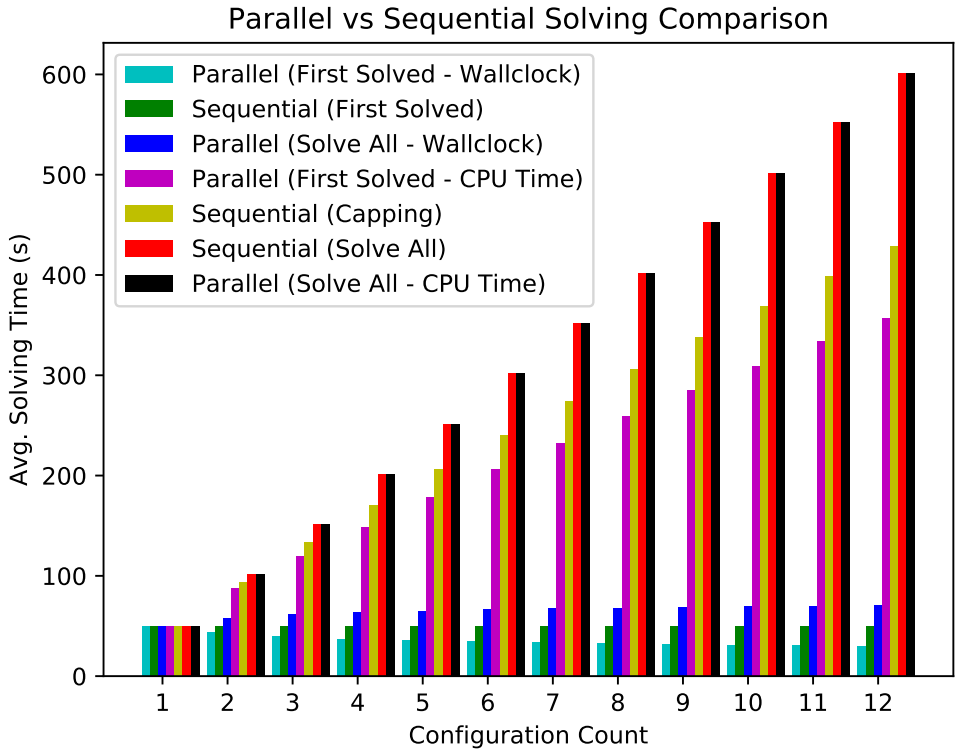


Figure 3.3: The average solving time of various solving approaches and configuration counts on a synthetic problem instance (over 1000 runs).

Figure 3.3 shows the results of these simulations. The bar chart shows the average solving time for a single instance (averaged over 1000 runs). We evaluate five different approaches, three sequential and one parallel. We present both "wall-clock time" (elapsed real time) and "CPU time" (the time elapsed across all CPU cores) for the parallel approach. In the sequential case these times are equivalent as a single core is used in all cases.

Parallel (First Solved) All configurations are run in parallel and terminated upon the earliest completion. This is the approach used in ReACT and provides partial information (best configuration only, all others are terminated and considered equal) on the configuration ranking with little overhead.

Parallel (Solve All) Configurations are run in parallel to completion allowing a complete ranking of each configuration evaluated on that instance to be created. Wall-clock time is determined by the longest running configuration (or cutoff), while CPU time is the summation over all configurations solving times (identical to its sequential counterpart).

Sequential (Brute-Force) All configurations are run to completion. This method takes the longest time to run but also provides the most information by giving the solving times and rankings for all participating configurations.

Sequential (First Solution) The run is terminated when any configuration finds a solution. While it is possible to quickly solve sequential instances using this method there are a number of drawbacks. No information is provided about the quality of configuration in relation to the others e.g. the worst configuration in the pool could be evaluated first and solve the problem. In cases where there is no timeout, as in this simulation, the first configuration evaluated in the sequence will always be used to solve the instance, this could lead to potentially very long solving times.

Sequential (Capped) Configurations are run sequentially with the current incumbent providing a tightening bound. Runs are terminated when the solving time exceeds that of the current incumbent. It provides a good balance between gaining information about configuration performance and solving time. This can be considered an aggressive form of the "adaptive capping" scheme introduced in [HHLBS09] as runs are terminated as soon as the incumbents solving time is exceeded, normally a multiple of the solving time is used. Adaptive capping is a common technique used in almost all state of the art algorithm configurators today [HHLBS09, HHL11, CLIHS17].

Figure 3.3 shows that in terms of wall-clock time the parallel approach (cyan) has the fastest solving time in all scenarios utilising two or more cores. Intuitively this makes sense as the parallel approach will always solve the instance using the fastest configuration amongst those evaluated. It is also worth noting that the solving time for this approach continues to decrease as the number of configurations evaluated increases. As more configurations are tested the likelihood of finding a better configuration

increase, this in turn will lead to a faster run termination.

The parallel approach trades fast wall-clock solving time for increased CPU computation. The parallel approach CPU solving time (pink) shows a modest improvement over the sequential capping approach (yellow) on average. Analogous to the wall-clock results, the parallel CPU solving time improves as additional configurations are tested. It is also worth noting that while the mean solving time for these two approaches is comparable the sequential approach has a large variance which is absent in the parallel case. The CPU time for the parallel approach can be calculated exactly as $t_b \times n$, where t_b is the solving time of the best configuration and n is the number of configurations run in parallel. The sequential capping approach has the same solving time in the best case (best configuration evaluated first) but $\sum_{i=0}^{n} t_n$ in the worst case (configurations evaluated in descending order of configuration time).

A first solution approach (green) provides the fastest mean solving time amongst the sequential solvers. There are however a number of drawbacks to this approach the most critical being that it cannot be used to perform algorithm configuration as no information about relative performance is gained. In addition to this, the solving time is decided entirely by which configuration is evaluated first and the cutoff time. In fact, this toy problem is very forgiving for this approach as all configurations are solvable between 0 and 100 seconds and selected from a normal distribution. For this reason the mean solving time for this approach (over 1000 runs) is equal to the mean sample distribution, 50 seconds. If the instances where generated more unforgivingly, such as sampled from a heavy-tailed distribution with no upper bound on solving time, this approach would suffer as slow configurations would eclipse better performing ones.

The longest solving time overall is as a result of the brute-force solving approach. The CPU time required is the same for both sequential (red) and parallel (black), though the wallclock time required is improved by parallelisation (blue). Allowing every instance is run until completion (either cutoff time or solving) allows the configurator to gain the most information about every configurations ranking relative to one another. This information comes at the cost of a large slowdown in overall solving time. This additional cost is difficult to justify as the additional information gained is about poor configurations and so unhelpful for steering the search towards promising areas of the search space (though model based configurators can likely use this information to train more accurate models). The drastic improvement observed in the Iterated F-Race algorithm with the addition of adaptive capping further supports this theory [CLIHS17].

It is clear from the above example that when solving instances with the objective of tuning a combinatorial solver there is a trade off between solving time and information

gain. On one extreme, approaches that expend only enough computing effort to solve the instance at hand provide too little information to conduct any configuration. On the other extreme methodologies that fully evaluate every instance to completion are often too slow to be practically useful. For this reason most prominent algorithm configurators adopt a compromise; capping the runtime by the best known solving time. If a parallel architecture is employed this capping can be even more aggressive while still proving information to allow configuration to occur. This aggressive capping approach uses only the resources required to solve the problem quickly while also gaining some information about the ranking of the configurations.

3.2.3 Configuration Pool and Leaderboard

In the previous example the information gained from each run is limited, as only the winning configuration is determined; the finishing order of other solver configurations are unknown due to early termination. The situation is further complicated by the fact that individual solver runs exhibit large fluctuations in solving times on the same instances due to stochastic behaviour such as randomised branching [HO15]. Despite these challenges it is still possible to use information from preceding races to surface good solver configurations. Utilising tested quality configurations in future races allows us to provide a bound on the worst-case performance as a solver run will not take longer than the solving time of the current incumbent.

Example 3.2.2 extends the previous example so that the winning solver configuration is part of the pool for future instances. While this a simplified synthetic example, it demonstrates the core idea of ReACT's approach: parallel racing and persistence of good configurations. Chapter 5 will take a more in depth look at advanced ranking strategies and selection methods aimed at nullifying the effect of stochasticity and noise in real world solving runs.

Example 3.2.2. Example 3.2.1 explored how various parallel and sequential approaches to solving performed on a single instance. Here we extend this to examine the effect of having the best configuration persist between instances. Configurations are generated in the same way as example 3.2.1 but now we solve 100 of the synthetic instances in a row. The configuration with the best solving time is included in the pool of configurations used to solve the next instance. We run this experiment 1000 times and show the mean solving time for the entire run in Figure 3.4. For clarity, only the most viable approaches from Example 3.2.1 are examined, namely "Parallel (First Solved)" and "Sequential (Capped)". A brute-force approach where all instances are run to completion is included as a baseline (this is equivalent to "Parallel (Solve All - Wallclock)" or "Sequential

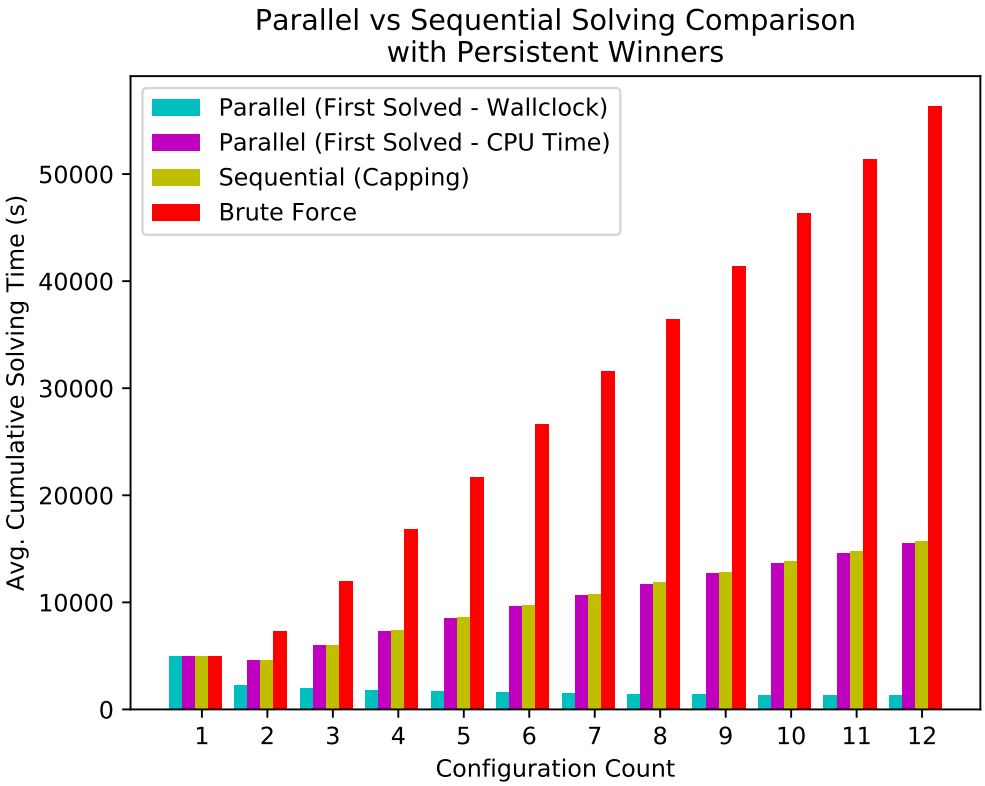


Figure 3.4: The average cumulative solving time of various solving approaches and configuration counts on a run of synthetic problem instances with the winning configurations persisting between instances.

(Solve All)" in Example 3.2.1).

Figure 3.4 show the results of these experiments. The most striking result is how the combination of two primitive implementations of parallel racing and winner persistence are able to drastically reduce the wallclock time required for the parallel approach (cyan). Notice also that this technique continues to improve as additional cores are added thus increasing the likelihood of finding an improving configuration due to the larger portfolio being run. The addition of winner persistence has closed the gap between the CPU time required for "Parallel (First Solved)" (pink) and "Sequential (Capped)" (yellow) with the former showing only the slightest of leads over the latter. The reason for this marginal lead is that the parallel approach is able to discover the best configuration in a run as soon as it finds a solution to an instance. On the other hand, the sequential approach must run the current incumbent first and then evaluate all other candidates using the incumbent time for capping until the new best is discovered. Figure 3.5 provides a visual representation of this. Looking closely at Figure 3.4 we see that the divide between approaches grows as additional cores are utilised. Intuitively this stands to reason; as the number of cores grow, the probability of the "new best"

running earlier in the sequential run drops. Another factor that would cause this bound to grow is if improving candidate configurations are included in the run frequently. Such a situation would arise when the configuration generation procedure is particularly strong. Finally, and unsurprisingly, we see that the brute-force approach (red) gains very little from keeping the incumbent from the previous races around (though there is a marginal improvement as while $n-1$ configurations are selected at random and run to completion, one configuration is constantly improving due to winner persistence).

Example 3.2.2 shows the impact that keeping a strongly performing configuration for future evaluations can have. The naive approach in Examples 3.2.1 and 3.2.2 can be dramatically improved by sampling from a fixed set of high quality configurations. We see in Algorithm 7 this configuration pool can be seeded with a set of known good quality configurations (for example the solver defaults) with the remainder of the allotted pool size filled in with configurations sampled at random from the configuration space.

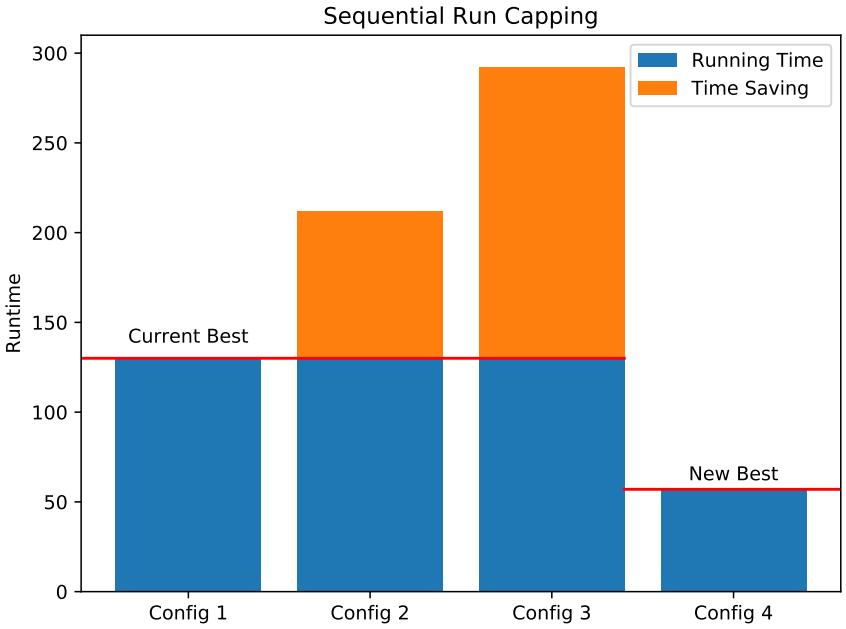
In the ReACT framework we limit the size of the pool. This is a necessity for two reasons. Firstly, from a practical point of view, allowing an extremely large or continuously growing configuration pool will consume excessive amounts of system resources such as memory. This is undesirable as one of the primary design principles of this ReACT framework is to keep the configurator footprint as small as possible. Secondly, the larger the pool the more difficult it is to evaluate each configuration thoroughly. For example, we are able to randomly sample six configuration at a time from the pool, a configuration pool of size 30 will take just 13 evaluations to reach a 0.95 probability of sampling every configuration. Extending the size of the pool to a much larger 250 configurations increases the required number of evaluations to 120 to reach the same probability. The formula for calculating the probability that a given instance is selected is given as:

$$1 - P(NotSelected)$$

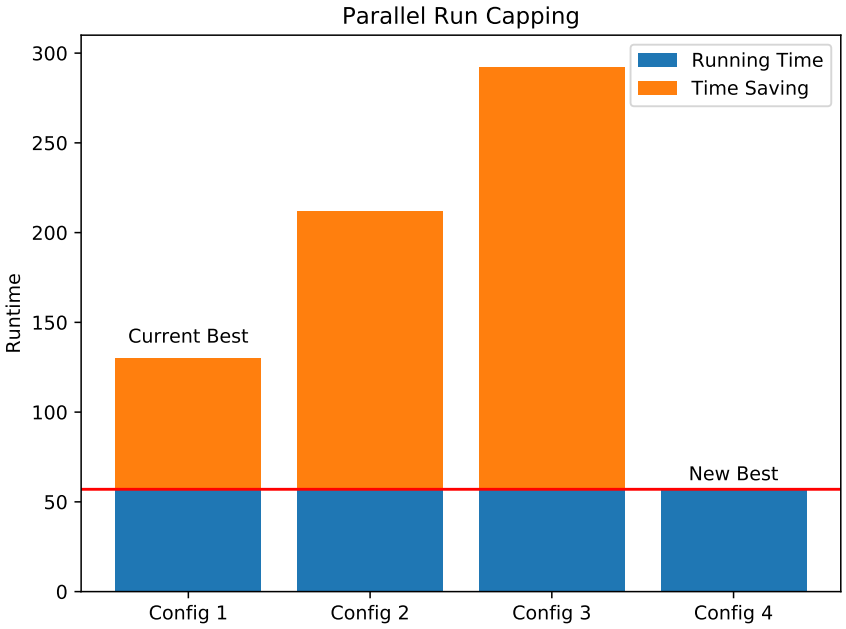
where

$$P(NotSelected) = (1 - \frac{ParallelRuns}{PoolSize})^{Evals}$$

From this it is apparent that the ratio of parallel evaluations to the configuration pool size drives the probability of configurations being sampled. When more configurations are run in parallel a larger configuration pool will work better. We see that when a configuration pool is large enough, the probability of an individual configuration being selected in any one run drops so low that an excessive number of configuration



(a) Sequential Runtime Capping



(b) Parallel Runtime Capping

Figure 3.5: A visual representation as to why parallel runtime capping outperforms sequential runtime capping.

races are required to reliably sample it. Because of this, it becomes difficult to collect enough performance data on the configuration to assess its strength relative to other configurations and to build a robust and stable leaderboard. Algorithm 7 shows us that the leaderboard is integral to many of the other processes which drive the ReACT system such as candidate selection, configuration removal, and configuration generation. For this reason it is imperative that the leaderboard be as accurate as possible.

3.2.4 Candidate Selection

With the configuration pool and leaderboard in place we turn our attention to the candidate selection procedure. This function (Line 15 of Algorithm 7) is responsible for selecting which configurations to run on an incoming instance from the configuration pool.

The candidate selection procedure often works in tandem with the leaderboard. The selection procedure uses the information from the leaderboard to select strong configurations which are likely to result in fast solving times. However, the relationship between these components works both ways. In order to improve the quality of the ranking in the leaderboard the selection procedure must strike a balance between using known good configurations and testing configurations that the system knows less about. Consistently using only reliable high quality configurations will lead to good performance but fails to explore the configuration space and so stunt improvement. On the other hand, only using new and untested configurations will explore the configuration space rapidly at the expense of the runtime performance. Luckily, as the ReACT system is parallel by nature it is possible to get the best of both worlds. To do so we must find a blend of configurations recognised to be good, in order to provide a bound on solving time, and unproven configurations with the potential to surpass the incumbent and improve the entire pool. Finding such a blend has a number of subtleties and also depends on the properties of the instance stream being processed. In Chapter 5 we investigate various blending methods to more fully understand how best to perform candidate selection.

Finally, it is also worth noting that in certain cases, such as the ReACT concrete instantiation described in Section 4.2, the selection procedure can be quite basic. Here, we are able to run all configurations due to the small configuration pool size. In these cases the candidate selection procedure has no work to do, instead simply returning all configurations.

3.2.5 Pool Maintenance

To allow the ReACT configurator to explore the configuration space effectively the configuration pool must not be allowed to stagnate. This is achieved through culling underperforming configurations and replacing them with new and potentially improving candidates. We refer to the combination of removal and generation as pool maintenance. Pool maintenance in Algorithm 7 is handled by the functions REMOVE CONFIGURATIONS and GENERATE CONFIGURATIONS (Lines 18 and 19). Pool maintenance is a nuanced subject where both the removal and generation procedure must strike balances. Chapter 6 covers the intricacies of pool maintenance in depth. However to place pool maintenance in the broader context of the framework we will give a brief synopsis of the challenges faced when adding and removing configurations.

Removing configurations from the configuration pool allows an equal number of new configurations to be added. The more new configurations that are added to the configuration pool the faster we are able to explore the configuration space and the greater the likelihood of discovering improvement. As such, ideally the removal procedure would rapidly remove any and all configurations which are known to not provide a benefit in solving instances. It is important that in the process of quickly removing underperforming configurations that the system does not also inadvertently cull a good configuration. Establishing the quality of a configuration requires evaluating the configuration on multiple instances. In general the more evaluations performed the higher our confidence in the assessment of that configuration. This leads to a trade off between speed and confidence. It is possible to quickly remove configurations after a small number of evaluations, however to do so runs the risk of unwittingly also removing a configuration which can provide value to the solving process. On the other hand, we can carefully assess the caliber of a configuration through many evaluations but this lowers the turnover in the configuration pool and slows the search for improving configurations. Striking a balance between these competing factors is the key to a powerful removal procedure.

Removing weak candidates is only part of the pool maintenance system, the other part is generating new potentially stronger configurations to fill the void left by removals. We can generate configurations that are minor variations of existing well performing configurations in the pool which exhibit similar good performance. The issue with solely focusing on this intensification approach is that it fails to adequately explore the larger configuration space and leads to becoming trapped in local optima. On the other extreme generating vastly different candidate configurations (through random sampling or other means) explores more of the configuration space but fails to exploit information gained

from previous runs. It is also improbable to discover good configurations without using some method of driving the search towards promising areas of the configuration space. Therefore robust generation procedures interleave both intensification and diversification to discover good configurations and then further improve them. The literature describes a number of different ways to achieve this goal [HHLBS09, AST09b, HHL11]. The design of the ReACT framework allows for the easy implementation of most approaches in the GENERATE CONFIGURATIONS procedure with the only caveat being that the computation cost does not impact the real-time nature of the configurator.

3.3 Chapter Summary

In this chapter we demonstrated that there is a real and practical need for the real-time algorithm configuration system outlined in this thesis. We detailed how, with the proliferation of multi-core systems, improving a combinatorial solvers configuration (parameter settings) while processing a stream of instances with no increase in wallclock solving time is now possible. An overview of our proposed novel framework for real-time algorithm configuration, ReACT, was presented along with explanations of the operations and interactions of the various component parts.

Chapter 4

Instantiations of the ReACT Framework

Summary. *In this chapter we provide two concrete instantiations of the ReACT framework, ReACT and ReACTR. We motivate the design decisions and important implementation details for both instantiations. We also present a full empirical evaluation of both instantiations which show them to be competitive with, or even exceed, the performance of the current state-of-the-art in offline algorithm configuration.*

The work presented here has appeared in the peer-reviewed conferences AAAI 2014 [FOMT14], SoCS 2014 [FMOT14] and IJCAI 2015 [FMO15].

4.1 Introduction

In Chapter 3 we presented the Real-time Algorithm Configuration through Tournaments (ReACT) framework. In this chapter we build on this work by implementing two concrete instantiations of the framework, ReACT and ReACTR. For each component in the framework we discuss and justify the implementation details chosen. In addition to this we outline the results of experiments conducted to evaluate each instantiation against both the solver defaults and configuration using a state-of-the-art offline configurator. Section 4.2 describes our first implementation of the ReACT framework, ReACT, and shows that despite its simplicity it is able to achieve strong results. ReACTR which improves on this initial work by adding a robust ranking system is described in Section 4.3. This section also shows ReACTR to be on a par with or exceed the

performance of existing state-of-the-art offline configurators across multiple domains and solvers.

4.2 ReACT: Real-time Algorithm Configuration through Tournaments

4.2.1 Overview

The first ReACT framework instantiation we will look at is ReACT [FMOT14]. ReACT was the first incarnation of the framework and was created with the purpose of exploring to what extent the ideas of the ReACT framework could work. Though many of the approaches used in this instantiation are simple they serve to show how the overall concept of the framework performs. We show that despite some naive design choices it is still possible to achieve performance exceeding that of the default solver configurations and approaching that achieved by state-of-the-art offline configurators. To begin we describe how each of the black box components in Algorithm 7 is realised. After that we evaluate the system and provide numerical results.

4.2.2 Leaderboard and Selection

Initialise Leaderboard Algorithm 8 shows the leaderboard and selection procedure implementations utilised in ReACT. To keep track of the performance of the configurations, we use a score keeping leaderboard L. This leaderboard is represented by the $n \times n$ matrix which is created by the function INITIALISE LEADERBOARD (Algorithm 8, Line 1). An entry $L(c_1, c_2)$ represents the number of times configuration c_1 has had a lower solving time than c_2 (assuming a solving time minimisation objective). Here solving time refers to the time taken to reach a solution that is within a user specified tolerance of the optimal in the case of optimisation problems or any satisfying solution in the case of satisfaction problems.

Candidate Selection With the leaderboard set up it is time to solve incoming instances. Recall from Algorithm 7 that for each instance in the instance sequence, ReACT uses the function SELECT CANDIDATES to decide which configurations should race against one another. In this instantiation of the algorithm, because the configuration pool size matches the number of parallel runs, SELECT CANDIDATES is simply a wrapper that returns all the configurations in the configuration pool (Algorithm 8, Line 6).

Algorithm 8 ReACT - Leaderboard and Selection

1: **function** INITIALISE LEADERBOARD(CP)
2: **Input:** CP, The current configuration pool.
3: **Output:** L, The initialised leaderboard for tracking configuration performance.
4: $L \leftarrow 0^{n \times n}$ ▷ an $n \times n$ matrix initialized to 0
return L
5: **end function**
6: **function** SELECT CANDIDATES(CP, L, R_n)
7: **Input:** CP, The current configuration pool.
8: L, The leaderboard for tracking configuration performance (not used).
9: R_n, The number of configurations to be run in parallel (not used).
10: **Output:** C, The list of configurations to run.
11: **return** CP
12: **end function**
13: **function** UPDATE LEADERBOARD(R, L, C)
14: **Input:** R, The runtime and solution status of the race.
15: L, The leaderboard for tracking configuration performance.
16: C, The configurations which ran in this race.
17: **Output:** L, The updated configuration leaderboard.
18: **for** configuration c_1 in C **do**
19: **for** configuration c_2 in C; $c_1 \neq c_2$ **do**
20: **if** $R[c_1][status] = Solved$ and $R[c_2][status] \neq Solved$ **then**
21: $L(c_1, c_2) \leftarrow L(c_1, c_2) + 1$
22: **end if**
23: **end for**
24: **end for**
return L
25: **end function**

Leaderboard Update After the selected candidates have finished racing against one another the race results are passed to the UPDATE LEADERBOARD procedure to perform a leaderboard update (Algorithm 8, Line 13). The procedure iterates over all pairs of configurations, the value of the cell in the corresponding matrix scoreboard is incremented in cases where one configuration triumphs over another. All other matrix cells remain unedited.

4.2.3 Pool Maintenance

Algorithm 9 ReACT Instantiation - Pool Maintenance

1: **function** REMOVE CONFIGURATIONS(L, m, r)
2: **Input:** L, The leaderboard for tracking configuration performance.
3: m, The minimum number of wins needed before configuration removal.
4: r, The win-loss ratio required before a configuration is removed.

5: **Output:** C_{remove}, A list of configurations to remove from the configuration pool.

6: $C_{remove} \leftarrow \emptyset$
7: **for** configuration c_1 in C **do**
8: **for** configuration c_2 in C; $c_1 \neq c_2$ **do**
9: **if** $L(c_1, c_2) \geq m$ and $\frac{L(c_1,c_2)}{L(c_2,c_1)} \geq r$ **then**
10: $C_{remove} \leftarrow C_{remove} \cup c_2$
11: **end if**
12: **end for**
13: **end for**
return C_{remove}
14: **end function**

15: **function** GENERATE CONFIGURATIONS(L, CP, Θ, L_n)
16: **Input:** L, The leaderboard for tracking configuration performance (not used).
17: CP, The current configuration pool.
18: Θ, The parameter configuration space of the solver algorithm A.
19: L_n, The number of configurations held in the leaderboard.

20: **Output:** C_{add}, A list of new configurations to add to the configuration pool.

21: $C_{add} \leftarrow SampleRandomConfigurations(L_n - |CP|, \Theta)$
return C_{add}
22: **end function**

4.2.3.1 Configuration Removal

After the scores are updated for the winners of the tournament, we cull *weak* configurations from the pool using the REMOVE CONFIGURATIONS method (Algorithm 9, Line 1). This method introduces two variables, m and r, which are specific to this instantiation of the framework. m, the minimum number of "wins" necessary to exclude a configuration, and r, the parameter weakness ratio.

We define a configuration c_1 as weak if it has: *(a)* been beaten by another configuration, c_2, at least m times and *(b)* the ratio of the number of times c_2 has beaten c_1 to the

number of times c_1 has beaten c_2 is greater or equal to r. The former criterion ensures that configurations are not removed without being given ample opportunity to succeed. The latter criterion ensures that the domination of one configuration over another is not just due to random chance. In this implementation of ReACT, we set $m = 10$ and $r = 2$, meaning a configuration is weak if it has lost to another competitor at least 10 times and has lost to another parameter setting twice as many times as it has beaten that parameter setting. For example, if c_1 beats c_2 ten times, but c_2 beats c_1 twenty times, then c_1 is a weak configuration and will be removed.

Although the approach is relatively straight forward, the presented scoring component does provide a reasonable guarantee that once a new improving configuration is found that it will retain those settings. In particular, for every time one new configuration defeats another, the latter must solve two additional instances to prove its dominance over the first. This means that any new candidate must prove its merit only a few times before it is taken seriously and becomes hard to remove by the current incumbent. Yet the requirement to solve twice as many instances simultaneously guarantees that it is possible for a new configuration to come along that does not have to spend a large number of runs to prove it is superior. This property allows the configuration to quickly adapt to changes in the observed instances while mitigating the chances we will get rid of a good configuration on a whim.

4.2.3.2 Configuration Generation

In striking a balance between intensification and diversification in its generation strategy, this ReACT instantiation tends towards diversity. Dominated configurations are replaced by those sampled at random from the configuration space (Algorithm 9, line15). The randomly sampled parameterisations provide strong diversification to avoid local optima. At first glance, simply choosing new configurations completely at random may seem too simple to actually function. However, there is a good reason for this randomness. As the instances change over time, diversity amongst parameters is critical to being able to find new configurations for those instances (this idea is discussed further in Chapter 5).

Here intensification is achieved by replacing poorly performing configurations. This strategy is still quite conservative, which stands in contrast to train-once parameter tuners like GGA, which intensifies through a strict selection procedure, or ParamILS, which has a greedy iterative first improvement step. ReACT benefits from keeping parameter settings that are "good enough" around. These configurations could turn out to be beneficial if the instances shift in a particular direction. By keeping such

parameters alive, our approach is always prepared for whatever may come next from the instance sequence. We only remove configurations when it is clear that they are dominated by other configurations, although it is possible that we are sometimes unlucky and throw out a good configuration.

One of the main drawbacks of this strategy, however, is that finding good configurations by chance is not easy. We are aided by the fact that there tend to be regions of good configurations, rather than single parameter configurations that perform extremely well; the downside of this approach is that by taking only one point from this region rather than systematically exploring it we may miss better configurations. By maintaining diversity, ReACT can hold on to parameters from a number of good regions and is prepared for new types of instances. In Section 4.3 we will show how more reasoned generation procedures lead to large performance improvements.

4.2.4 Experimental Setup and Datasets

In order to properly test ReACT, we require a dataset with several properties. First, the instances must be relatively homogeneous, i.e. the instances are all of a similar type or variant of a problem. Without homogeneity, we cannot assume that a single parameter set can work well across all of the instances. Heterogeneous datasets are best handled using per-instance algorithm configuration and algorithm selection techniques which can tailor a solution to the diverse instances present in a heterogeneous dataset. Second, the instances must be hard, but not too hard. Instances requiring too much time to solve will timeout, thereby offering no information to an algorithm configurator about which parameter setting is best. This results in the configurator performing a random walk. Meanwhile, datasets that are too easy will not provide noticeable gains.

We use two distinct datasets of combinatorial auctions problems encoded as mixed integer problems that we generated using the Combinatorial Auction Test Suite (CATS) [LBPS00a]. Using CATS, we were able to create datasets consisting of a homogeneous set of instances that are solvable in a reasonable amount of time. CATS has a number of different options for generating problem instances, including the number of bidders and goods in each instance, as well as the type of combinatorial auction.

We focused on two types of auction distributions (regions and arbitrary) in order to provide homogeneous datasets for our tuner to work with. The regions problem class is meant to simulate situations where proximity in space matters. This is typically the case when plots of land are auctioned, where developers typically seek adjacent plots of

land for developments. Alternatively, the arbitrary class is designed so that there is not a determined relation between various goods, as is likely to be the case when dealing with, for example, collector's items. While both datasets deal with combinatorial auctions, the structures of the corresponding instances are very different and therefore lend themselves to different solution strategies. The reader is referred to the original CATS algorithm description [LBPS00a] for more information about the instances.

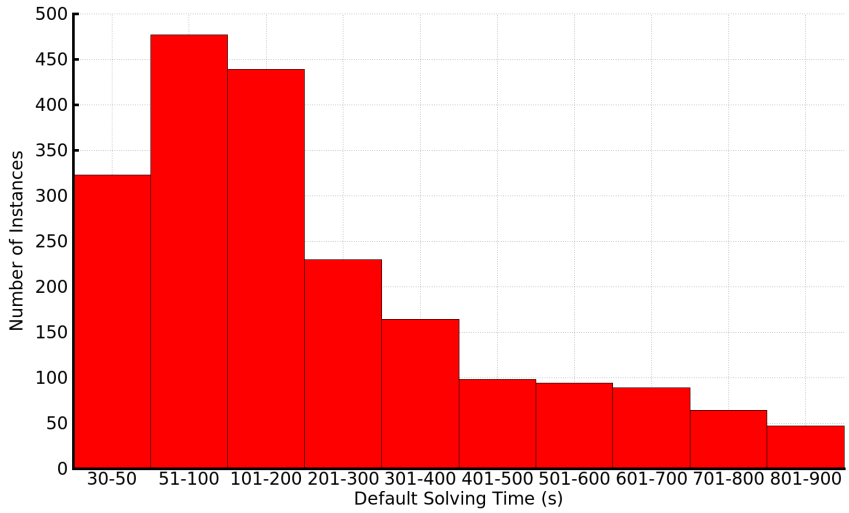
For the regions dataset, we generate our instances with 250 goods (standard deviation 100) and 2000 bids (standard deviation 2000). The instances we generated can be solved in under 900 seconds using the default solver configuration. Combining this with a solver timeout of 500 seconds means that some of the instances are challenging, but can be potentially solved with slightly better configurations within the time limit. The arbitrary instances were generated with 800 goods (standard deviation 400) and 400 bids (standard deviation of 200). These experiments maintained the 500 second timeout.

We solve the instances in our dataset with IBM CPLEX [IBM14]. CPLEX is a state-of-the-art mathematical programming solver used widely both in industry and academia. The solver has over 100 adjustable parameters that govern its behaviour, and has been shown in the past to be rather amiable to automated configuration [KMST10, HHL11]. Specifically, we use the version of CPLEX in algorithm configuration benchmarking library ACLib, namely version 12.6.1. ACLib is commonly used in the literature and provides a configuration space description for CPLEX that exposes 74 parameters.

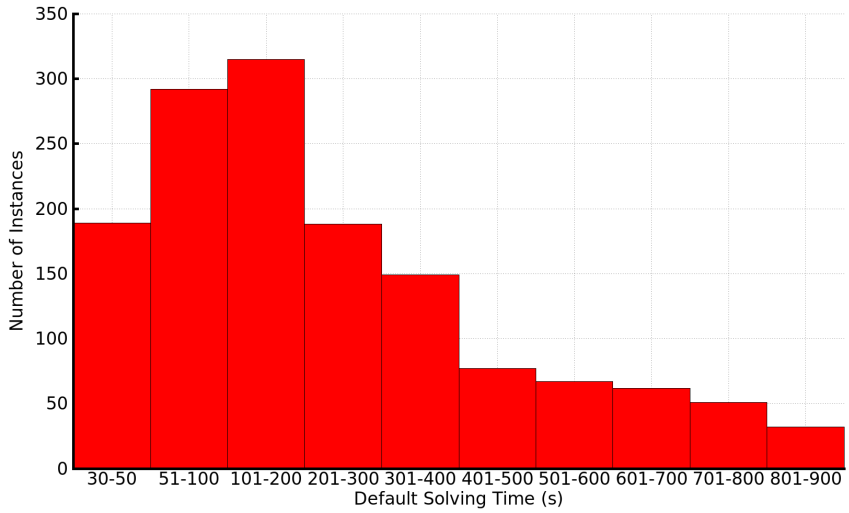
When generating our benchmark dataset, we removed all instances that could be solved in under 30 seconds using the default CPLEX parameter settings. These instances are too easy to solve and would likely introduce noise into our configuration procedure due to system timing variances. We stress that removing these instances can be justified in the presence of a straightforward pre-solver, a practice commonly used for algorithm selection techniques prior to predicting the best solver for an instance. Our final regions dataset is comprised of 2,000 instances whose difficulty, based on performance of the default parameters, was mainly distributed between 30 and 700 seconds, but with all instances being solvable in under 900 seconds, as shown by Figure 4.1a. Meanwhile, the final arbitrary dataset comprises of 1,422 instances, whose distribution of runtimes with default parameters can be seen in Figure 4.1b.

4.2.5 Experimental Evaluation

We test our methodology on three different scenarios on each of the two combinatorial auction datasets. In the first, we assume that instances are being processed at random,



(a) Regions



(b) Arbitrary

Figure 4.1: The CPLEX default configuration solving time distribution for both regions and arbitrary combinatorial auctions datasets.

so we shuffle all of our data and feed it to our tuner one at a time. We also try two variations where problems change steadily over time. In the first case, we assume that the auction house grows and is able to take in larger inventories, so the number of goods in each new instance is monotonically increasing. In the second case, we create a scenario where the auction house becomes more successful, thus each new problem instances has a monotonically increasing number of bids. Regardless of the scenario, however, each instance is given a timeout of 500 seconds on a 2X Intel Xeon E5430 Processors (2.66Ghz) with 8 cores. We restrict ourselves to using only 6 cores so as to

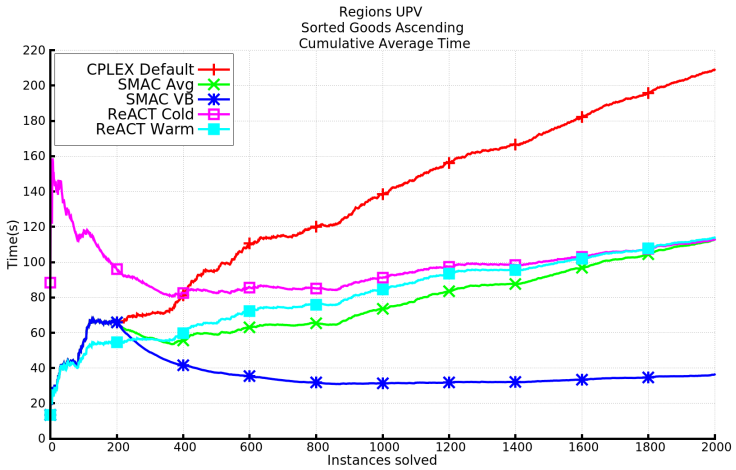
leave some overhead for system processes in order to avoid any impact on timings.

As a comparison, in addition to the default parameters, we compare ReACT with the state-of-the-art train-once approach SMAC [HHL11]. Here we simulate the case where initially no training instances exist and we must use the default parameters to solve the first 200 instances. SMAC then uses the first 200 instances as its training set and is tuned for a total of two days. After those 48 hours, each new instance is solved with the tuned parameters. In all presented results we follow the generally accepted practice of tuning multiple versions with SMAC and selecting the best one based on the training performance. In our case we present both the result of the best found SMAC parameters, and the average performance of the six configurations we trained. We refer to these as SMAC-VB and SMAC-Avg, respectively. Note that in this case SMAC-AVG is the performance one would expect from running SMAC on a single core. SMAC-VB on the other hand is equivalent to running all 6 tuned versions of SMAC in parallel using the same number of cores as are available to ReACT.

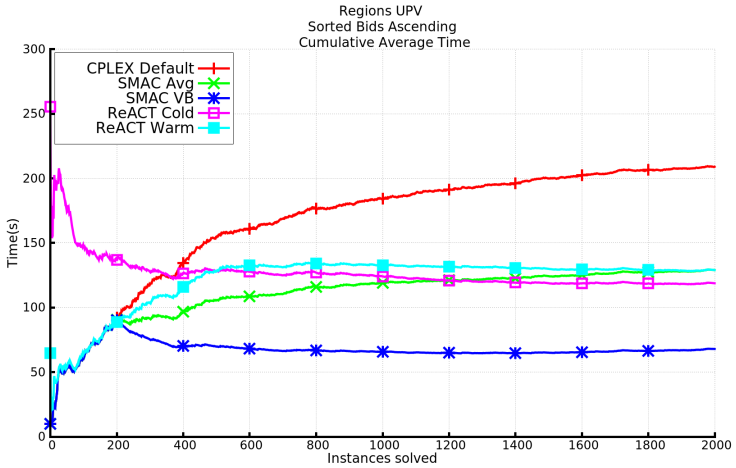
For our evaluations, we compare two versions of ReACT. In the first, we assume that no information is known about the solver beforehand, and thus all the initial configurations are generated at random. We refer to this case as ReACT-cold. Additionally, we also test ReACT-warm, where one of the initial configurations contains the default settings.

To avoid presenting a result due to a lucky starting configurations or random seed, we run each version of ReACT three times, and present the average of all three runs in the plots and tables of this section. Figures 4.2 and 4.3 summarise the results on the regions datasets. First note that Figure 4.2a presents the cumulative average time per instance for the scenario, where the number of goods continuously increases with each instance. Note that the default curve rises steadily with each new instance. This is because as the number of goods increases, the difficulty of the problem also increases, and the default parameters are unable to adapt. Alternatively, notice that ReACT-warm, which initially has the default parameters as one of its starting parameters, is able to adjust to the change and achieve a significant improvement after only 150 instances. Even when the initial parameters are chosen at random, ReACT-cold is able to quickly achieve a level of performance such that the cumulative average outperforms the default in only 400 instances.

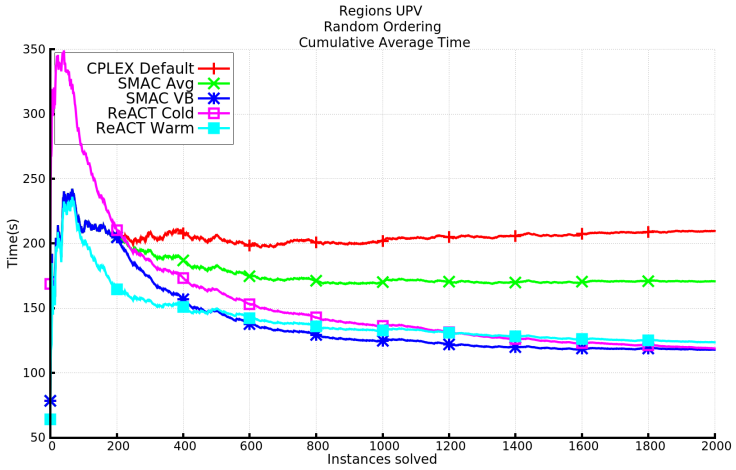
Figures 4.2b and 4.2c tell a similar story. In each case within observing 200 to 400 instances either version of ReACT is able to overtake the performance of default CPLEX parameters. The fact that ReACT comes so close to the performance of SMAC with such a simple approach, in which the next configuration is selected completely at random, shows the power of the core ReACT framework ideas: parallel racing and aggressive



(a) Dataset with number of goods monotonically increasing.



(b) Dataset with number of bids monotonically increasing.



(c) Dataset with random ordering of instances.

Figure 4.2: Cumulative average runtime of techniques on the three permutations of the regions dataset. The x-axis specifies the total number of observed instances. The y-axis specifies the solution time in seconds.

capping.

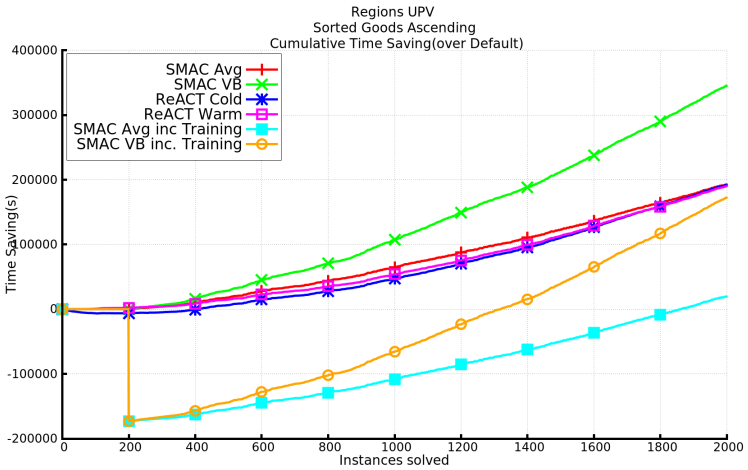
The true benefit of ReACT is visible clearly in Figures 4.3a, 4.3b, and 4.3c. These figures present the cumulative amount of time saved by the associated methodology over using the default parameters. After observing the corresponding cumulative average time, it is no surprise that the savings continuously improve with each newly observed instance. What we also observe at first glance is that the parameters found by ReACT perform as well as those for a typical result after a 2-day SMAC run. ReACT seems not that much worse than the SMAC-VB. This observation, however, is somewhat misleading. Recall that in order to find the SMAC parameters, a 2-day training period is essential. During this training time a company would have been forced to use the default parameters, which do not scale well. If we consider that on average instances are coming in continuously, in the time it took to train SMAC, another 800 instances would have gone by. However, with ReACT, good parameters are found throughout the testing period. In fact, we are able to definitively tune over 70 parameters in real-time with minimal overhead and achieve a performance drastically better than default. We also achieve a performance on par with the average SMAC run.

Furthermore, let's hypothetically assume that after we observe the first 200 instances, that no new instances arrive for the next two days. This is an unlikely scenario, but it allows us to definitively show the computational cost of the training in Figures 4.3a, 4.3b, and 4.3c in the form of SMAC VB-inc and SMAC Avg-inc. Here, we clearly see that even after the remaining 1800 observed instances, the parameters found by SMAC are not able to offset the upfront cost of finding them. Meanwhile ReACT, is able to find its parameters with no temporal overhead.

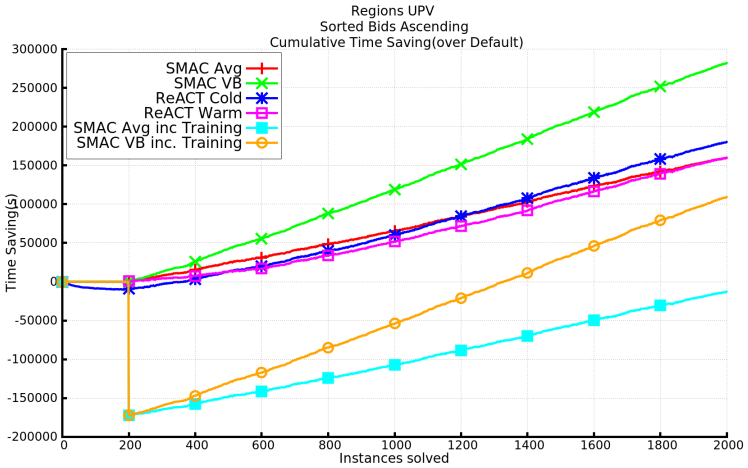
Note also that this means that although SMAC VB inc appears to steadily converge on ReACT, it may not be indicative of the long term behavior. ReACT continuously discovers better parameters as it progresses and is likely to eventually find parameters better than those of SMAC while the parameters which SMAC uses remain unchanged.

Additionally, it cannot be said that SMAC VB is not using as many resources as ReACT. We tune several versions of the solver in parallel and use the lowest per-instance solving time in our results. Thus, SMAC VB utilises the same number of cores as ReACT.

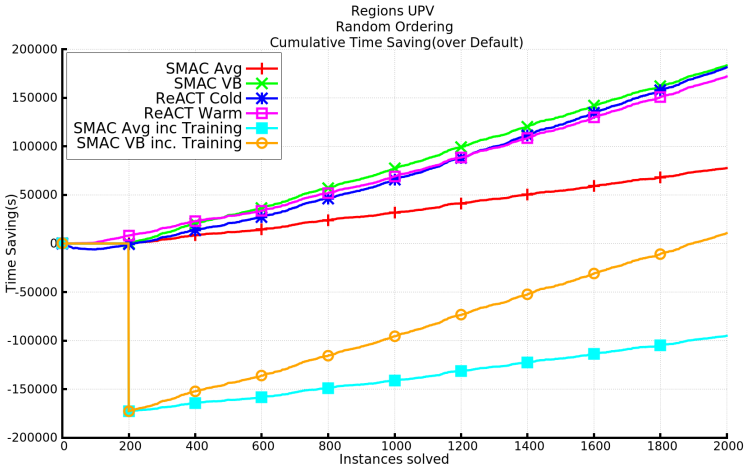
Figures 4.4 and 4.5 show the plots on arbitrary auctions for a random ordering of incoming instances and also scenarios where the number of goods and bids increase with each subsequent instance. Note that for this dataset the cold-started ReACT is slower in finding a parameter set that outperforms the default, yet even on these harder instances the gains become evident. And just like in the regions scenarios, the upfront



(a) Dataset with number of goods monotonically increasing.

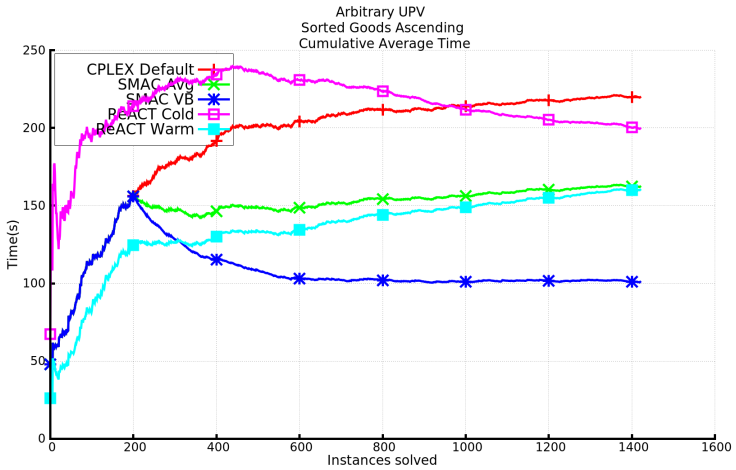


(b) Dataset with number of bids monotonically increasing.

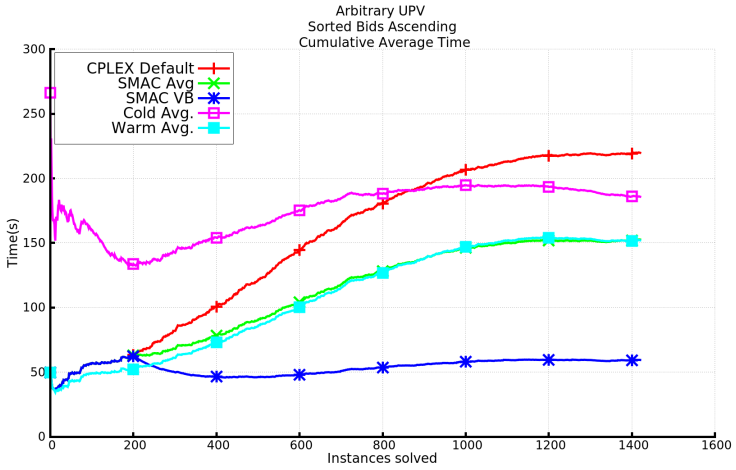


(c) Dataset with random ordering of instances.

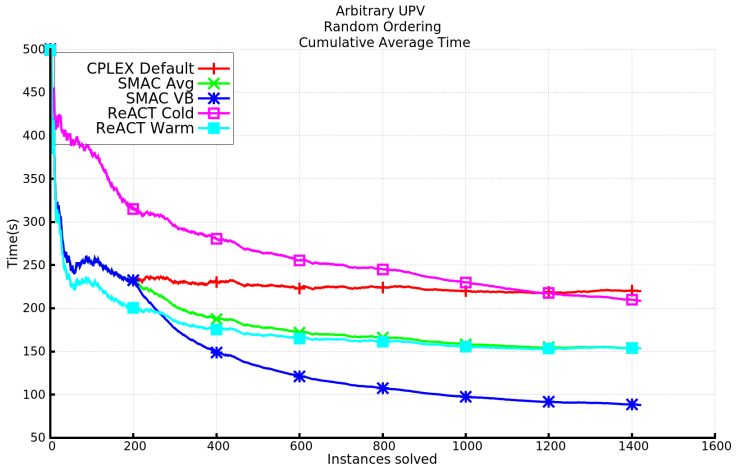
Figure 4.3: Cumulative time savings of techniques on the three permutations of the regions dataset. The x-axis specifies the total number of observed instances. The y-axis is the number of seconds saved over the default configuration of CPLEX.



(a) Dataset with number of goods monotonically increasing.

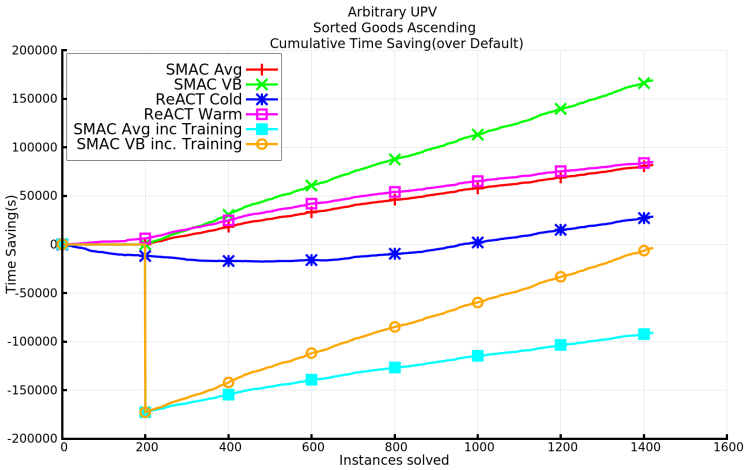


(b) Dataset with number of bids monotonically increasing.

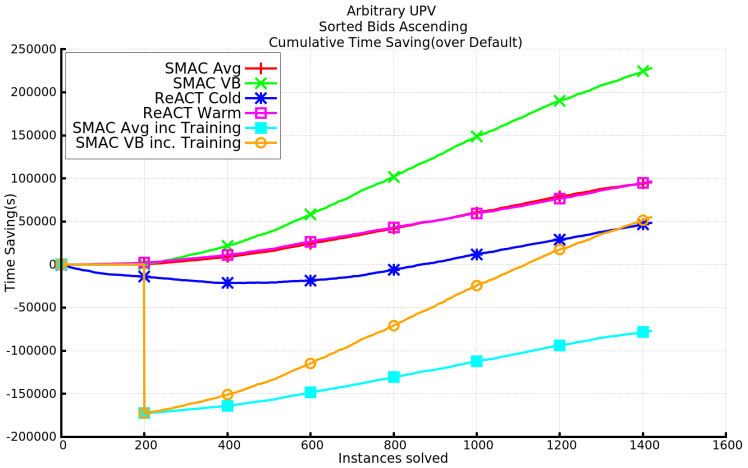


(c) Dataset with random ordering of instances.

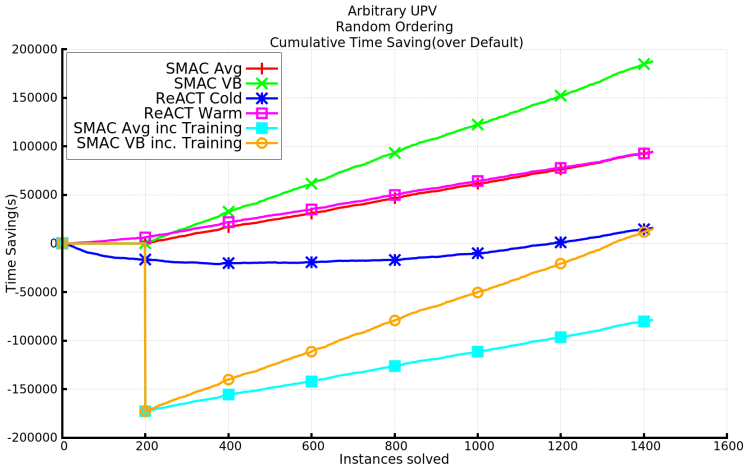
Figure 4.4: Cumulative average runtime of techniques on the arbitrary dataset. The x-axis specifies the total number of observed instances. The y-axis specifies the time in seconds.



(a) Dataset with number of goods monotonically increasing.



(b) Dataset with number of bids monotonically increasing.



(c) Dataset with random ordering of instances.

Figure 4.5: Cumulative savings of techniques on the arbitrary dataset. The x-axis specifies the total number of observed instances. The y-axis is the number of seconds saved over the default configuration of CPLEX.

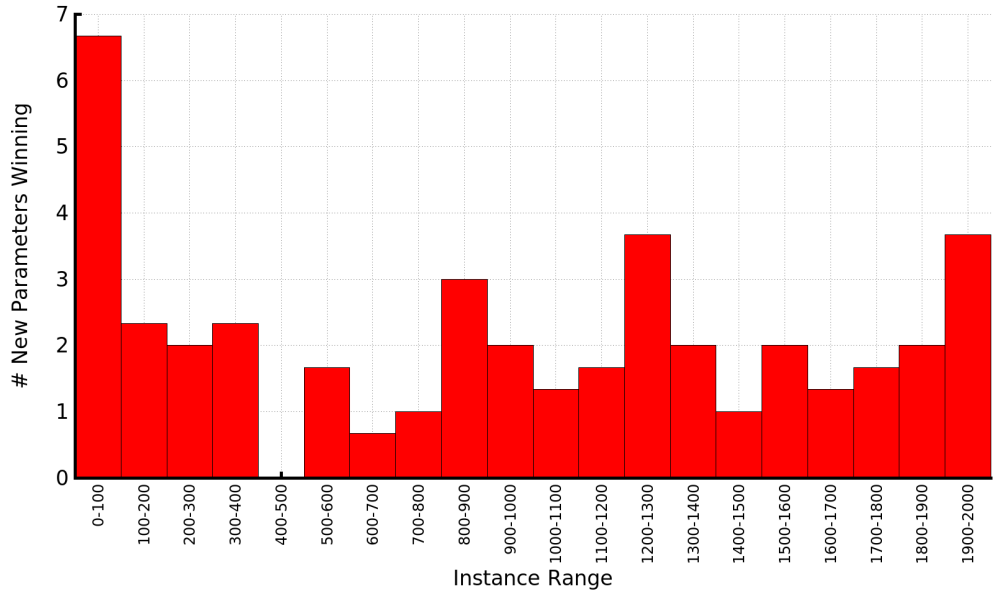


Figure 4.6: Histogram showing the number of new winning configurations appearing per 100 instances solved on the Regions dataset (sorted goods ascending).

tuning costs necessitated by SMAC are hard to overcome.

As an alternative view of the results, Table 4.1 presents the amount of cumulative time saved by each tuning approach over the default parameters. The numbers are in thousands of seconds. In all cases, ReACT is able to outperform default parameters and is on a par with SMAC Avg. Given the fact that the CPLEX solver has been hand tuned by experts for over a decade and SMAC is the current state-of-the-art, this is an encouraging feat. Interestingly, for the case where the new regions instances arrive in a random order, ReACT overtakes the performance of the average SMAC parameter set and is on close to that of SMAC VB. This is likely because the first 200 randomly sorted instances tend to be harder in general, so two days might not be enough to fully tune SMAC. However, ReACT is not affected by this as it does not require initial offline tuning.

We show one last piece of experimental evidence in order to show that this instantiation of the ReACT framework does not just find a good configuration and remain using it for the entire configuration run. This competition is important as a dominant incumbent could lead to stagnation in the configuration pool. Using cold-started ReACT on the Regions dataset (sorted by ascending number of goods) we investigate how often newly introduced configurations are successful. The histogram in Figure 4.6 shows the average number of times a new configuration wins a tournament in a discretised time period over the solution horizon. Unsurprisingly, our method has a significant amount of

Table 4.1: Cumulative amount of time saved over the default CPLEX parameters in thousands of seconds after n instances have been solved. Negative values indicate additional solving time. Three orderings of the regions and arbitrary datasets are shown: random, in which instances are processed in a random order; sorted goods, where the number of goods in each subsequent instance is monotonically increasing; and sorted bids, in which the number of bids of each new instance is monotonically increasing.

Regions Random	After n Instances			
	500	1000	1500	2000
SMAC Avg.	12	31	54	78
SMAC Virtual Best	29	77	131	183
SMAC Virtual Best (inc.)	-144	-96	-42	11
ReACT Cold-start	21	66	122	181
ReACT Warm-start	28	69	118	172

Arbitrary Random	After n Instances			
	350	700	1050	1400
SMAC Avg.	12	39	65	93
SMAC Virtual Best	24	78	130	185
SMAC Virtual Best (inc.)	-149	-95	-43	12
ReACT Cold-start	-20	-17	-7	15
ReACT Warm-start	17	43	68	93

Regions Goods Sorted (Asc)	After n Instances			
	500	1000	1500	2000
SMAC Avg.	18	65	123	192
SMAC Virtual Best	29	106	212	345
SMAC Virtual Best (inc.)	-144	-66	39	172
ReACT Cold-start	6	47	110	192
ReACT Warm-start	15	54	113	190

Arbitrary Goods Sorted (Asc)	After n Instances			
	350	700	1050	1400
SMAC Avg.	13	40	60	81
SMAC Virtual Best	22	74	120	166
SMAC Virtual Best (inc.)	-151	-98	-53	-7
ReACT Cold-start	-17	-14	5	27
ReACT Warm-start	20	48	68	84

Regions Bids Sorted (Asc)	After n Instances			
	500	1000	1500	2000
SMAC Avg.	24	65	113	160
SMAC Virtual Best	42	118	202	282
SMAC Virtual Best (inc.)	-130	-54	30	109
ReACT Cold-start	13	60	121	180
ReACT Warm-start	13	52	105	160

Arbitrary Bids Sorted (Asc)	After n Instances			
	350	700	1050	1400
SMAC Avg.	6	33	64	94
SMAC Virtual Best	15	79	158	224
SMAC Virtual Best (inc.)	-158	-93	-15	51
ReACT Cold-start	-20	-14	16	46
ReACT Warm-start	9	34	63	95

"churn" as it begins to sort through poor configurations at the beginning of its execution. However, this short period of tumult ends fairly quickly, with ReACT finding a few new configurations in each subsequent 100 instance period. Of important note is that new configurations are winning right up to the end of the period of optimisation, exactly as we would expect.

It is interesting to note that, while we continuously observe new potential configurations that occasionally win the tournament, in all our scenarios there is usually one parameter set that the solver continuously comes back to two thirds of the time. This means that our approach is able to quickly find a good configuration and use it as a new core, while occasionally taking advantage of some configuration getting lucky. Here the parallel racing mechanism of the ReACT framework behaves like a parallel portfolio approach by selecting a strong pool of potential candidates to compete in parallel on an instance. The fact that no other configuration is able to kick this core one out, is also a testament that the scoring metric we use is fair yet resilient to noise. This is again confirmed by stating that for warm start ReACT, in all observed cases, the default parameters are thrown out after at most 300 instances.

4.3 ReACTR: Real-time Algorithm Configuration through Tournament Ranking

4.3.1 Overview

Although straightforward in its implementation, our first instantiation of the ReACT framework showed the potential of real-time algorithm configuration. ReACT was able to very quickly find high quality configurations in real-time while still return problem solutions in as little time as possible. We will now turn our attention to another instantiation of the ReACT framework, Realtime Algorithm Configuration through Tournament Rankings (ReACTR), which introduces enhancements to every part of the original. Specifically, we expand the configuration pool and invoke a far more involved candidate selection procedure to decide which configurations should be evaluated once a new instance arrives. We also improve how and which candidate configurations should be introduced into the pool of potentials to be evaluated. Most importantly, we show how a ranking scheme commonly utilised to rank players in games, TrueSkill, can be exploited to more accurately measure the quality of the configurations in the current pool.

The primary reason for ReACT's success was that it always ran the best configuration,

and had a very aggressive removal policy, throwing out anything as soon as it was even hinted to be subpar. The trouble with this strategy, however, was that it failed to store a history of candidate configuration successes. Due to this, a new configuration could potentially remove a tried and tested candidate simply by getting a few lucky runs when it was first added to the pool. Therefore, it should be clear that the success of the ReACT methodology resides in the strategies used for each of the steps. Particularly important is the consideration of which configurations to evaluate next (SELECT CANDIDATES), which configurations should be discarded (REMOVE CONFIGURATIONS), and how new configurations should be added to the pool (GENERATE CONFIGURATIONS). This section targets each of these questions, showing how employing a powerful leaderboard to rank the current pool facilitates all other decisions.

4.3.2 Leaderboard and Selection

Leaderboard At the core of the enhancements made to produce the ReACTR instantiation of the ReACT framework is a strong and stable method of comparing the relative merit of the configurations under consideration. To this end we seek inspiration from the competitive games. In the world of competitive games, it is critical to have an unbiased way to compare an individual's performance to everyone else. A trivial way to do this is to simply have everyone play everyone else. Naturally, for popular games like chess, Go, checkers, Halo, Counter-Strike etc., there are a plethora of people playing at any given time, with players coming and going from the rankings on a whim. This is almost identical to the situation that ReACT faces internally. At any given time there are a number of competitors in the pool, which can be removed and replaced by new entrants based on their relative strength. At the same time, ReACT needs a method for quickly determining the comparative quality of each of the contestants in the pool, to know the ones worth utilising and those that can be discarded. It therefore makes sense to employ a leaderboard ranking algorithm, such as those commonly utilised for board-game and video-game ranking.

The de facto standard for ranking systems is the Elo rating system and its many extensions [Elo78]. For brevity we will leave a full discussion of the various ranking systems to Chapter 5. However, it suffices to know that the Elo predictor introduces the idea that the point difference in two players' ratings should correspond to a probability predictor on the outcome of a match.

The issue with most Elo based ranking systems is that they are primarily designed for two player games. Therefore, for games involving 3+ players, all combinations of pairs must be created and updated independently in order for classic Elo approaches to work.

To solve this multi-player problem for online games, the Bayesian ranking algorithm TrueSkill [HMG06] was invented. TrueSkill measures both a player's average skill, μ, as well as the degree of uncertainty (standard deviation), σ, assuming a Gaussian distribution for a player's skill. TrueSkill uses Bayesian inference in order to rank players. After a tournament, all competitors are ranked based on the result, the mean skill is shifted based on the number of players below a player's rank and the number above, weighted by the difference in initial average ratings. Again we will leave discussion of specifics to a later chapter. In these experiments we use a highly-rated open source Python implementation of TrueSkill [Zon14].

The ability to accurately and quickly rank multiple configurations competing in each race lays a strong foundation for the other components in this ReACTR instantiation. Additionally, the confidence metric provided by TrueSkill is a highly desirable feature that helps the system to determine whether it is worth continuing to evaluate a configuration or whether it can be safely discarded. The improved ranking system also allows for an increase in the number of candidate configurations under consideration at any one time. For these evaluations we have increased the size of the leaderboard and configuration pool to thirty while maintaining the number of parallel evaluations at six.

To use TrueSkill with ReACTR we initialise our leaderboard as a mapping from configurations to TrueSkill scores (Algorithm 1, Lines 5 and 6). The TrueSkillScore function simply creates an object to track the skill (μ) and confidence deviation (σ). Initially we use the default values of $\mu = 25$ and $\sigma = 8.3\dot{3}$ for generated configurations and a marginally higher μ value for any configurations included by warm starting (this allows these configurations to be included in the initial selection).

At the conclusion of a race in ReACTR information about the competitors and the race results are passed to the TrueSkillScoreUpdate function to recalculate the skill and confidence for each configuration (Algorithm 1, Line26). TrueSkill uses Bayesian methods to calculate the updated scoring information while the exact calculations are performed by approximate message passing on a factor graph representation of the model. Chapter 5 discusses this in more detail and we refer the reader to [HMG06] for full details of the methods.

Candidate Selection Using TrueSkill for ranking allows each member in our current pool of thirty potential configurations (this can be set an arbitrary value) to have an associated score declaring its quality, as well as confidence rating of this score. To guarantee that the overall solver's performance will be as good as is currently possible, the best known configuration is always among those that is evaluated. We refer to the

Algorithm 10 ReACTR Instantiation - Leaderboard and Selection

function INITIALISE LEADERBOARD(CP)
    Input: CP, The configurations in the current configuration pool.

    Output: L, The initialised leaderboard for tracking configuration performance.

    $L \leftarrow \emptyset$
    for configuration c in CP **do**
        $L[c] \leftarrow TrueSkillScore()$
    end for
    return L
end function

function SELECT CANDIDATES(CP, L, R_n)
    Input: CP, The current configuration pool.
        L, The leaderboard for tracking configuration performance.
        R_n, The number of configurations to be run in parallel.
        ϵ, The portion of configurations used for exploration of the configuration space.

    Output: C, The list of configurations to run.

    $C \leftarrow \emptyset$
    $C = C \cup SelectBestTrueSkillScore(CP, L, R_n * 1 - \epsilon)$
    $C = C \cup SelectRandom(CP, L, R_n * \epsilon)$
    return C
end function

function UPDATE LEADERBOARD(R, L, C)
    Input: R, A vector which holds the runtime and solution status of the race.
        L, The leaderboard for tracking configuration performance.
        C, The configurations which ran in this race.

    Output: L, The updated configuration leaderboard.

    $ranks \leftarrow GetRanks(C, R)$ ▷ GetRanks returns a vector containing 1 for the race winner and 2 for all other competitors.
    $ts_{scores} \leftarrow TrueSkillScoreUpdate(C, ranks)$
    for configuration c in C **do**
        $L[c] \leftarrow ts_scores[c]$
    end for
    return L
end function

method used to select the configurations to run from the leaderboard as the sampling strategy or the candidate selection procedure.

We must always include the current best configuration in order to bound our runtime, and a number of sampled configurations so that we can explore the configuration space, our sampling method can be considered a parallel variation of the commonly used ϵ-greedy strategy. ϵ-greedy strategies are used in situations where there is a trade-off between exploration and exploitation. The strategy uses the best known option a portion of the time while randomly sampling other options the rest of the time. In our scenario we have the luxury of using multiple cores and so are able to dedicate a portion of these to running incumbents while simultaneously exploring with the other cores. In these experiments we chose, somewhat arbitrarily, to run the top two known solvers and the others chosen at random($\epsilon = 0.66\dot{6}$).

4.3.3 Pool Maintenance

4.3.3.1 Configuration Removal

Even with a high quality ranker, such as TrueSkill, if there are no good configurations in our pool of contenders, we will never be able to improve our overall performance. Therefore, we must decide when a configuration is considered inferior and how confident we need to be in this estimation. For these ReACTR experiments we empirically discover good threshold values to use.

Because the evaluation of ReACTR on real data requires us to run a solver for non-trivial amount of time over a large number of instances, finding the best strategy for removing instances can be extremely expensive. To overcome this, we simulate the process using synthetic data. Specifically, we know that because our instances are relatively homogeneous in practice, any configuration of a solver would have a particular expected performance with some variance. We further assume that there is a particular mean expected performance and a mean variance. Therefore, each configuration of a solver is simulated by a normal distribution random number generator with a fixed mean and standard deviation. Once this simulated solver is removed, it is replaced by another one, where the new mean and variance are assigned randomly according to a global normal random number generator. This means that most of our simulated solvers have a similar performance, with those being significantly better than others being increasingly unlikely.

Given these simulated solvers, the objective of ReACTR is to find a solver that leads to the shortest cumulative solving time for 500 instances. Of course since these solvers

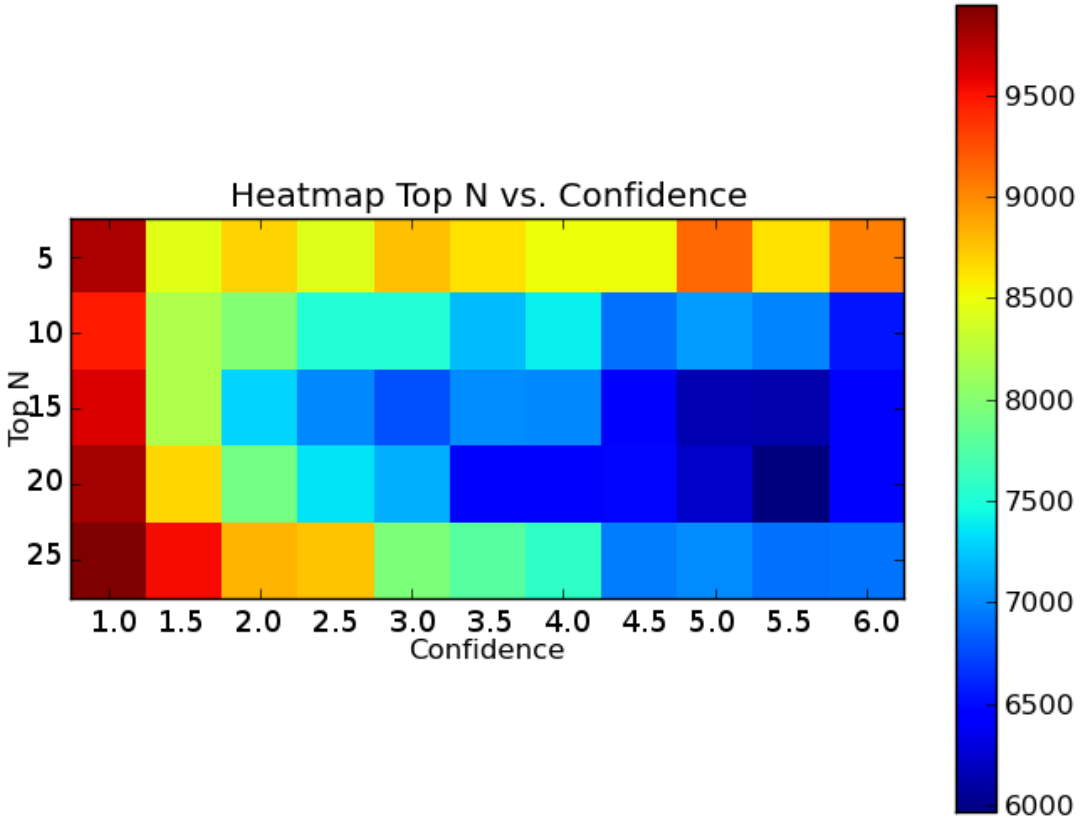


Figure 4.7: Heat-map showing the combined effect of the number of kept configurations and TrueSkill's confidence rating(lower values indicate higher confidence). The colourbar shows the cumulative solving time in seconds.

are random we repeat the experiment several times. What we observe is displayed in Figure 4.7. Specifically we observe that the best strategy for removing configurations from the pool of contenders is by keeping the top 15-20 configurations and anything in which we have a better than 5.0-5.5 uncertainty rating.

We use these values for our ReACTR experiments going forward removing anything that falls outside the top 15 configurations and has a confidence value of less than five (remember the confidence value represents a confidence interval, so a smaller value means more certainty). The function REMOVE CONFIGURATIONS in Algorithm 11 (Line 1) shows this in action. The μ_{thresh} value is set to that of the 15^{th} ranked configuration (outside of the function) while σ_{thresh} is hardcoded to the empirically derived value. The function then iterates over all configurations and adds those which fail to meet the criteria to the removal list. In all of the upcoming experiments that we have conducted using these emperical values performed filtering of the instances quite proficiently, however, it is a distinct possibility that these hardcoded values may not be optimal in all cases. For this reason, we investigate alternative and more adaptive

Algorithm 11 ReACTR Instantiation - Removal and Generation

function REMOVE CONFIGURATIONS(L, C, μ_{thresh}, σ_{thresh})
    Input: L, The leaderboard for tracking configuration performance.
        C, The list of configurations.
        μ_{thresh}, The TrueSkill score threshold.
        σ_{thresh}, The TrueSkill confidence required.
    Output: C_{remove}, The configurations to remove from the configuration pool.
    $C_{remove} \leftarrow \emptyset$
    for configuration, c in C **do**
        ▷ Lower TrueSkillConfidence score indicates the system is more confident.
        if $TrueSkillScore(c) \leq \mu_{thresh}$ **and** $TrueSkillConfidence(c) \leq \sigma_{thresh}$ **then**
            $C_{remove} \leftarrow C_{remove} \cup c$
        end if
    end for
    return C_{remove}
end function

function GENERATE CONFIGURATIONS(L, CP, Θ, L_n, I, n, m)
    Input: L, The leaderboard for tracking configuration performance.
        CP, The current configuration pool.
        Θ, The parameter configuration space of the solver algorithm A.
        L_n, The desired number of configurations.
        I, The ratio of exploration to exploitation.
        n, The number of top configurations to consider for breeding.
        m, The mutation probability.
    Output: C_{add}, A list of new configurations to add to the configuration pool.
    $C_{add} \leftarrow \emptyset$
    $N \leftarrow L_n - |CP|$
    for $i \leftarrow 1$ to N **do**
        if $Random() \leq I$ **then** ▷ Exploration
            $C_{add} \leftarrow C_{add} \cup SampleRandomConfigurations(1, \Theta)$
        else ▷ Exploitation
            $P_a \leftarrow SelectRandomFromBest(n, CP, L)$
            $P_b \leftarrow SelectRandomFromBest(n, CP, L)$
            $child \leftarrow Crossover(P_a, P_b)$
            if $Random() \leq m$ **then**
                $child \leftarrow Mutate(child)$
            end if
            $C_{add} \leftarrow C_{add} \cup child$
        end if
    end forreturn C_{add}
end function

solutions for instance removal in Chapter 6.

4.3.3.2 Configuration Generation

Every configuration which is removed from the leader-board must be replaced by a newly generated configuration. In order to balance exploration of new configurations and the exploitation of the knowledge we have already gained, ReACTR uses two different generation strategies. Our method for this is demonstrated in the GENERATE CONFIGURATIONS function of Algorithm 8. In the same way we did with the ReACT instantiation of the framework we ensure diversity by generating configurations where the value of each parameter is set to a random value from the range of allowed values for that parameter (Line 29).

Additionally, ReACTR exploits the knowledge it has already gained through ranking by using a crossover operation similar to that used in genetic algorithms. For this, two parents are chosen from amongst the highest ranked configurations then with equal probability each parameter takes one of the parents values (Lines 31, 32 and 33). For these experiments we decided, somewhat arbitrarily, upon the top 5 as being high ranking (controlled by the variable n in our pseudocode). Additionally, like a standard genetic algorithm, some small percentage of the parameters are allowed to mutate (Line 35). For our experiments we set the variable m, which controls mutation probability to 0.05. That is, in 5% of cases, rather than assuming one of the parent values, a random valid value is assigned instead.

We allow for a variable, I, to control the balance of exploitation to exploration, or in other words, the percentage of generated versus random configurations we introduce (Line 28). In the following experiments we generate exploration and exploitation configurations in equal proportion i.e. $I = 0.5$.

4.3.4 Experimental Setup and Datasets

In addition to the two combinatorial auction datasets used to assess the performance of ReACT, we evaluate ReACTR on six new SAT datasets. These datasets are selected from the configurable SAT solver challenge (CSSC) [HLB$^+$17]. CSSC 2014 ran four tracks: Industrial SAT+UNSAT, crafted SAT+UNSAT, Random SAT+UNSAT, and Random SAT. At least one dataset from each track in the competition is selected in order to represent a broad variety of SAT domains. In particular we choose competition datasets where configuration resulted in the largest improvement over default. We also take this opportunity to test ReACTR's effectiveness on the three winning solvers in CSSC 2014. Below we summarise these new solvers and datasets.

Table 4.2: Configurable SAT Solver Challenge Solvers Overview [HLB+17]

Solver	# Parameters				# Configurations		Reference
	c	i	r	cond.	original	discretised	
Lingeling	102	139	0	0	1×10^{974}	1×10^{136}	[Bie13]
ProbSAT	5	1	3	4	∞	1×10^{5}	[BS12]
Clasp-3.0.4-p8	38	30	7	55	∞	1×10^{49}	[GKNS07, GKS12a]

These ReACTR experiments where run on the same systems following the same methodology as previous ReACT experiments (describe in Section 4.2.5), as are all other experiments in this work, unless otherwise noted. We show the results for the SMAC VB scenario only as this matches the resources used by ReACTR and so is a fairer comparison. SMAC was tasked with optimising the PAR10 score which penalises timeout at ten times the cutoff value. In all cases we perform six runs of each configuration method and present the average result to account for variance and the stochastic nature of solvers.

4.3.4.1 Solvers

Lingeling is a CDCL SAT solver which ranked 1st in the industrial track of the CSSC 2013 [Bie13]. It also placed highly in the crafted SAT+UNSAT and random SAT+UNSAT (achieving 3rd and 2nd place respectively). In these experiments we use the 2013 version of Lingeling which exposes 241 parameters, the largest configuration space in the competition. The exact break down of the parameter types exposed is summarised in Table 4.2.

Clasp-3.0.4-p8 is a conflict-driven nogood learning answer set programming solver [GKNS07, GKS12a]. It also has the ability to solve (Max-)SAT and pseudo-Boolean problems. It won both the crafted SAT+UNSAT and random SAT+UNSAT tracks in both CSSC 2013 and CSSC 2014 (it also achieved 3rd in the CSSC 2014 industrial track). The SAT solving portion of Clasp has 75 parameters including 55 conditional options to control its search strategies (summarised in Table 4.2).

probSAT is described as "a pure and simple probability distribution based solver... probably one of the simplest SLS solvers ever presented" by the paper introducing it [BS12]. probSAT exposes only nine parameters which primarily control shape of the probability distribution. Despite its simplicity probSAT managed to achieve 1st place in the random SAT track of the CSSC 2014.

At this juncture it is worth noting that parallel versions of both Lingeling and Clasp

Table 4.3: Configurable SAT Solver Challenge Datasets Overview [HLB+17]

Benchmark	# Train	# Test	# Variables	# Clauses	Reference
Circuit Fuzz	299	585	5.53k±7.45k	18.8k±25.3k	[BLB10]
GI	1032	351	11.2k±17.8k	2.98m±8.03m	[Tor13, MB13]
N-Rooks	484	351	38.2k±37.4k	125k±126k	[MS14]
K3	300	250	262±43	1116±182	[BTH14]
3cnf	500	250	350±0	1493±0	[BY13]
5sat500	250	250	1000±0	4260±0	[TBH11]

exist [Bie10, GKS12b]. Similarly CPLEX is able to take advantage of multiple cores present in the system [IBM14]. Despite this, in all of these experiments we opt to use the single threaded versions provided in the Algorithm Configuration Library (ACLib) [HLF+14]. This is in keeping with the methodology used to evaluate previous parallel configuration approaches from the literature such as GGA(++) and distributed SMAC [AST09b, AMS+15, HHL12]. It should also be pointed out that there is no reason that ReACTR would be unable configure parallel solvers, however, given the resources required to evaluate this we leave this for future work.

4.3.4.2 Datasets

Circuit Fuzz For the Industrial track we use circuit fuzzing instances. This dataset was independently generated using FuzzSAT [BLB10]. FuzzSAT first generates a boolean circuit and then converts this to CNF. We solve these instances using the popular SAT solver Lingeling [Bie13]. The circuits were generated using the options `-i 100` and `-I 100`. Any instances that could be solved in under 1 second using the lingeling defaults were removed. The resulting dataset contained 884 instances which was split into 299 training and 585 test[1]. We use a time-out of 300 seconds for this dataset, the same as that used in the CSSC 2013.

Graph Isomorphism (GI) These instances in the crafted SAT+UNSAT track are created by encoding the graph isomorphism problem as a SAT problem [Tor13, MB13]. Due to this formulation the resulting instances typically contain a large number of clauses (see Table 4.3). The dataset we use consists of 1032 training instances and 351 test instances. We solve these instances using the Clasp solver with a time-out of 300 seconds.

[1]The number of test instances differs from the description in [HLB+17] but private correspondence with the authors confirms this to be the correct figure for CSSC 2013

N-Rooks Our second crafted SAT+UNSAT dataset considers a variant of the classic N-Queens problem where instead of placing n queens on an $n \times n$ checker board we must instead place a number of rooks in such a way that they cannot attack each other [MS14]. The instances in this particular dataset are unsatisfiable, requiring $n + 1$ rooks be placed on an $n \times n$ board. The generator is parametrised to allow rooks to be fixed to certain rows or columns thus making it easier to prove unsatisfiability. The dataset consists of 484 training instances and 351 test instances. These are again given a 300 second solving budget using Clasp.

K3 This dataset from the random SAT+UNSAT track is a collection of randomly generated 3-SAT instances generated using the random instance generator from the 2009 SAT competition [BTH14]. The instances were generated in batches of 100 with a variable count in the range 200 to 325 (increasing in increments of 25) and a corresponding clause count which places them at the phase transition (clause to variable ratio of approximately 4.26). There are 300 train and 250 test instances which we solve with Clasp given a 300 second time-out.

3CNF This is another set of randomly generated 3-SAT instances from the random SAT+UNSAT track of CSSC 2014. These instances are generated using ToughSAT with 350 variables and 1493 clauses so that they are at the phase transition [BY13]. Clasp with a 300 second cut-off to solve the 500 train and 250 test instances.

5SAT500 This dataset consists of 500 5-SAT instances (split evenly between train and test) from the random SAT track of CSSC 2014. The instances are generated to have 500 variables and 10000 clauses (a clause-to-variable ratio of 20) [TBH11]. Our solver of choice for these instances is the SLS solver probSAT given a solving budget of 300 seconds.

For a full description of all solvers and datasets see the paper outlining the CSSC [HLB$^+$17]. It is important to re-emphasise here, that ReACTR by its nature is an online algorithm, and therefore does not require a separate training dataset. However, in order to compare to existing methodologies, a training set is necessary for those approaches to work hence why the datasets are split into train and test groups.

4.3.5 Experimental Evaluation

This section outlines two sets of experiments conducted to evaluate ReACTR. The first set, published in the paper introducing ReACTR [FMO15], uses three scenarios: the

two combinatorial auctions datasets used to evaluate ReACT (regions and arbitrary, solved by CPLEX), and a set of circuit fuzzing instances from the CSSC 2013 (solved by Lingeling). Our second set of experiments expands on this work to show ReACTR's versatility using two new solvers and five new datasets from the CSSC 2014.

To begin with we look at this first set of experiments, where we show two separate scenarios for ReACTR. First, we consider a scenario where there is a collection of training data available beforehand, or alternatively a training period is allowed. We refer to this approach as "ReACTR Merged", where the configurator is allowed to make a single pass over the training instances to warm-start its pool of configurations. Secondly, we evaluate "ReACTR Test", which assumes no prior knowledge of the problem, and starts the configuration process only when it observes the first test instance. It is worth noting that this warm-start procedure differs from the warm-start outlined in the previous ReACT experiments hence the difference in terminology. We believe that warm-starting by including the solver defaults, as shown in the previous experiments, to be the most common scenario for the end user of ReACTR to encounter, and for this reason, all ReACT and ReACTR runs in this section seed their configuration pools with the solver default configuration.

For comparison we evaluate both versions of ReACTR against a state-of-the-art static configurator, SMAC [HHL11]. For completeness, we investigated SMAC when trained for 12, 24, 36 and 48 hours. This way we cover scenarios when a new solver is configured each night, as well as the best configuration SMAC can find in general. We observed that on our particular datasets performance improvements stagnated after 12 hours training, except in the case of the regions dataset (for which we show the 24 hour training). Furthermore, because ReACTR uses six cores, six versions of SMAC are trained using the 'shared model mode' option, which allows multiple SMAC runs to share information. Upon evaluation, all six configurations are run and the time of the fastest performing SMAC tuning on each instance is logged. By doing this the CPU time used by SMAC and ReACTR is comparable.

We also show the results for both the previous version of ReACT (on the merged dataset described above), "ReACT Merged" and the default solver configurations.

Figure 4.8 shows the rolling average (total time to date/instances processed) on the circuit fuzz dataset. We can see that both versions of ReACTR easily outperform the Lingeling defaults. What is more interesting is that ReACTR is able to outperform SMAC (trained for 12 hours) after a single pass over the training set (taking under 4 hours). Even without the warm-start, ReACTR is able to find parameters that are significantly better than the defaults and not too far off those that were otherwise

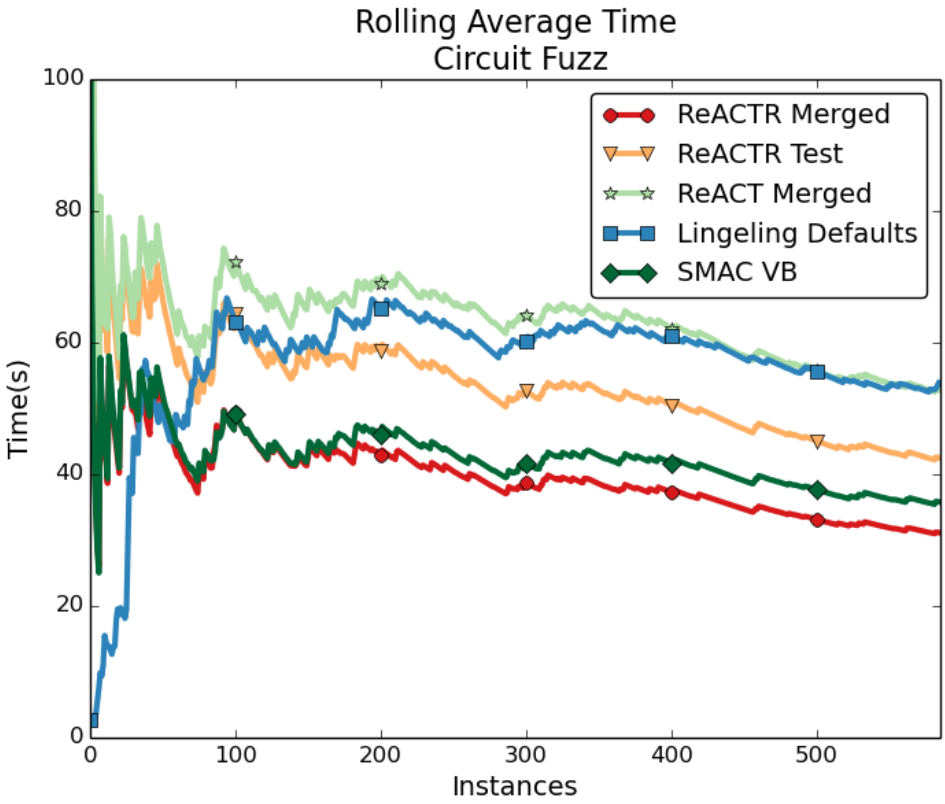


Figure 4.8: Lingeling Configuration: Circuit fuzzing dataset.

configured.

In Figure 4.9, we see that on the Arbitrary Combinatorial Auction dataset, configuration is extremely important, and that all configurators are able to find the good configurations. In Figure 4.10, however, we once again see that both versions of ReACTR find significantly better configurations than those that can be found after 12 hours of tuning SMAC, and even the configuration found after 24 hours of tuning SMAC.

Table 4.4 shows the amount of time each configuration technique requires to go through the entire process. This shows the amount of time needed to train the algorithm and the amount of time needed to go through each of the test instances. The times are presented in 1000s of seconds. Note that in all cases, ReACTR Merged requires less time to train and also finds better configurations than SMAC. However, if training time is a concern, then ReACTR Test requires significantly less total time than any other approach.

We now focus our attention on the second set of experiments, configuring the solvers Clasp-3.0.4-p8 and ProbSAT on a variety of benchmarks from the CSSC [HLB+17]. Given the larger number of datasets and the expense involved in running these configuration experiments we limit ourselves to a subset of evaluations run on the previous three benchmarks. Specifically we run the solver default configuration, ReACTR Merged,

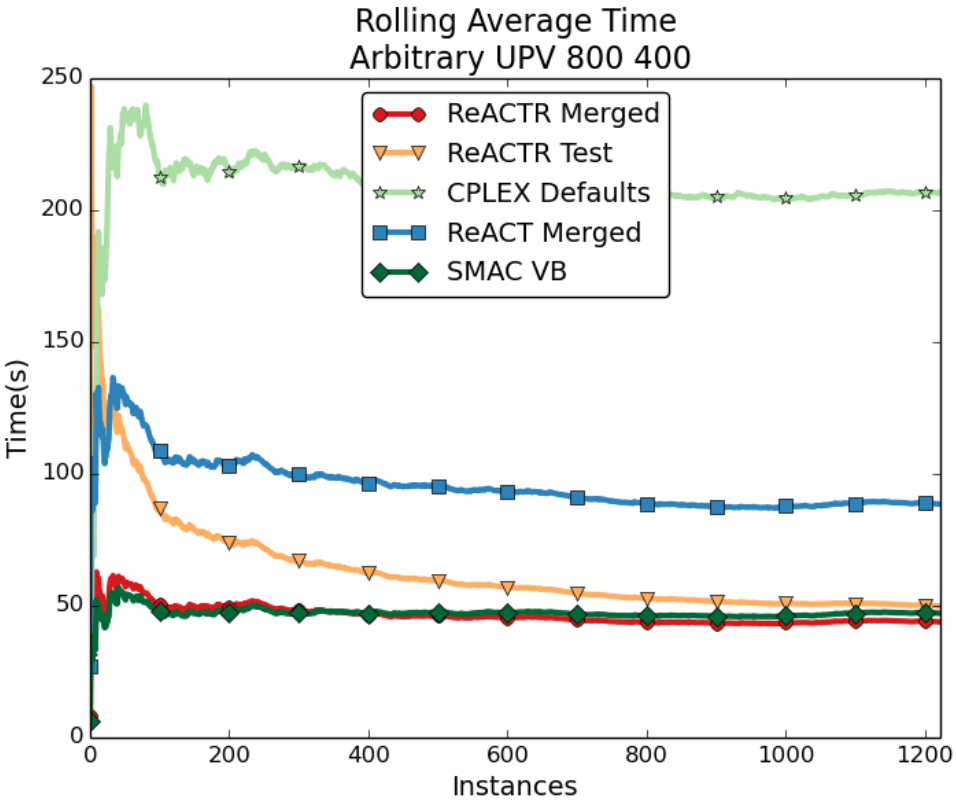


Figure 4.9: CPLEX Configuration: Arbitrary combinatorial auctions dataset.

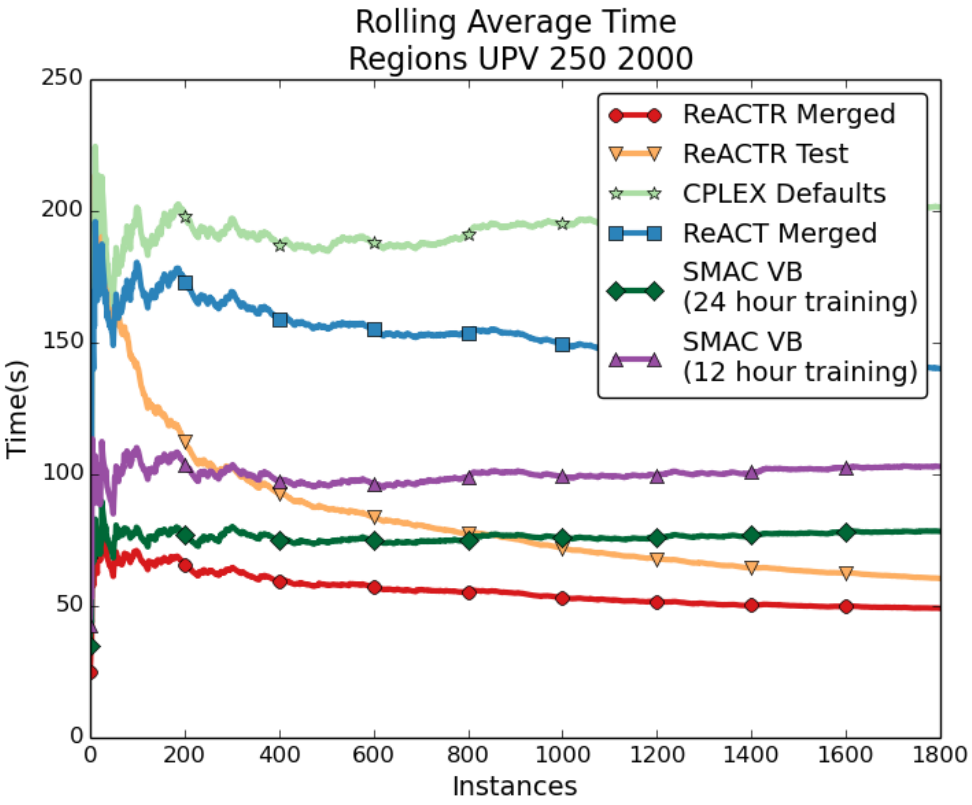


Figure 4.10: CPLEX Configuration: Regions combinatorial auctions dataset.

Table 4.4: Summary of training, testing and total time needed for the various configurations on the benchmark datasets.

		Time taken (1000s)						
		Solver	ReACTR		SMAC (number of hours)			
		Default	Test	Merged	12	24	36	48
Regions	Train	0	0	24	43	86	130	173
	Solve	363	109	**88**	185	141	110	102
	Total	363	**109**	112	228	227	240	275
Arbitrary	Train	0	0	19	43	86	130	173
	Solve	253	61	**54**	58	57	57	57
	Total	253	**61**	72	101	143	186	230
Circuit Fuzz	Train	0	0	13	43	86	130	173
	Solve	32	25	**18**	21	21	21	21
	Total	32	**25**	31	64	107	150	194

Table 4.5: Mean total solving time and number of time-outs for various scenarios from the CSSC 2014.

		Default		**ReACTR**		**SMAC**	
		Solve Time (s)	Timeouts	Solve Time (s)	Timeouts	Solve Time (s)	Timeouts
clasp-3.0.4-p8 3cnf-v350	Train	0	*NA*	33577	*NA*	172800	*NA*
	Test	32068	33	11651	0	9725	0
	Total	**32068**	33	45228	**0**	182525	**0**
clasp-3.0.4-p8 gi	Train	0	*NA*	40153	*NA*	172800	*NA*
	Test	14391	43	12878	35.5	9003	$15.66\dot{6}$
	Total	**14391**	43	53030	35.5	181803	$\mathbf{15.66\dot{6}}$
clasp-3.0.4-p8 K3	Train	0	*NA*	2771	*NA*	172800	*NA*
	Test	2306	0	1049	0	719	0
	Total	**2306**	**0**	3820	**0**	173519	**0**
clasp-3.0.4-p8 queens	Train	0	*NA*	17940	*NA*	172800	*NA*
	Test	29551	$82.8\dot{3}$	8831	$10.8\dot{3}$	2365	0
	Total	29551	$82.8\dot{3}$	**26771**	$10.8\dot{3}$	175165	**0**
probSAT 5SAT500	Train	0	*NA*	447	*NA*	172800	*NA*
	Test	75000	250	294	0	124	0
	Total	75000	250	**741**	**0**	172924	**0**

and SMAC with 48 hours training (the default scenario supplied by ACLib, where all new test scenarios are sourced [HLF+14]).

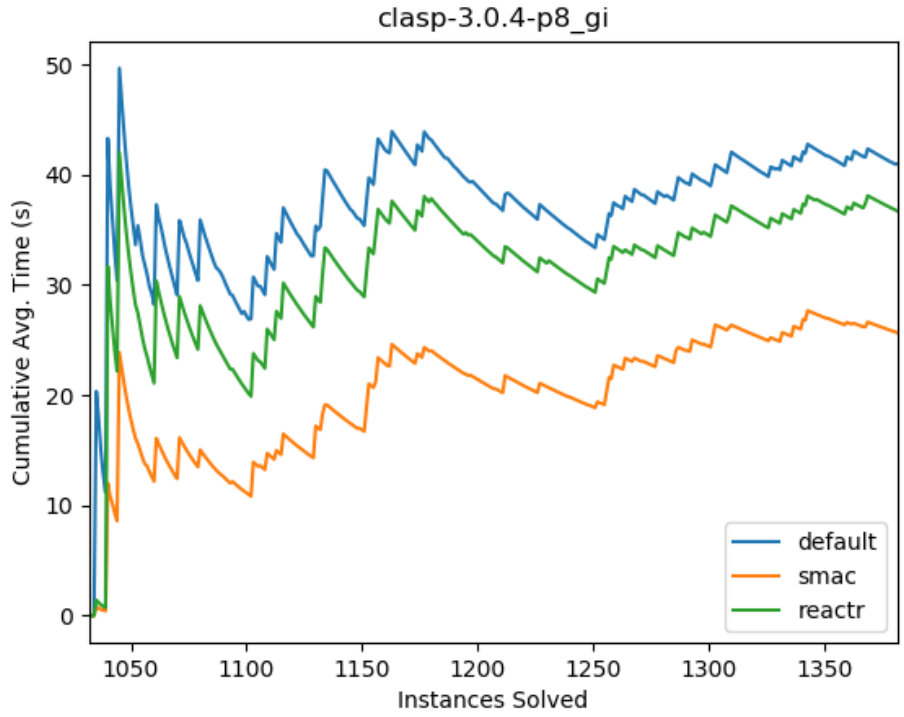
Looking at Table 4.5 we see that in this set of experiments there is a clear ranking both in terms of overall solving time and the number of instances which time-out. In all cases SMAC achieves the lowest total solving time and number of time-outs on the test set. ReACTR also always improves over the performance of the default configuration

on the test set but to a lesser extent.

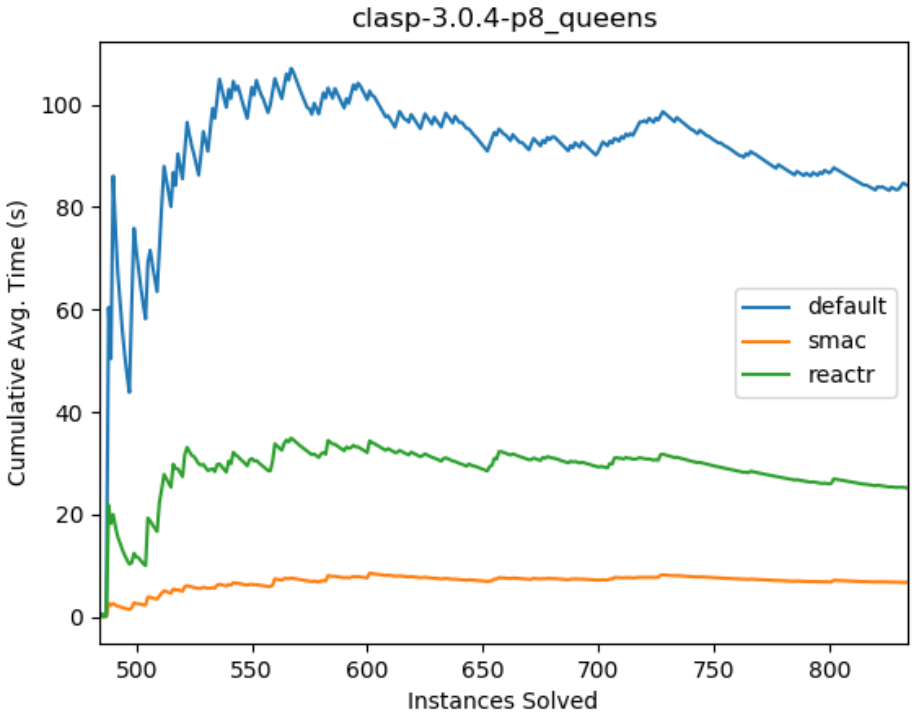
Considering the overall configuration time rather than just the time taken to solve the test instances shifts these results considerably. In three of the five benchmarks no configuration would be the preferable option (in terms of solving time)! On the remaining two benchmarks, N-Rooks and 5SAT500, ReACTR achieves the lowest overall solving time despite including the time to perform a single pass over the training set (which is not strictly necessary, but does result in a better configuration). The results on the 5SAT500 dataset are particularly impressive; a single pass over training set, taking only 447 seconds, allowed all test instances to be solved where previously all timed out.

There are however two important caveats to these results. Firstly, if eliminating time-outs is important in the use case then foregoing configuration is only viable in a single case, the K3 benchmark. Configuration using ReACTR would be preferable for the 3CNF and 5SAT500 datasets while SMAC would be the superior choice for both the GI and N-Rooks benchmarks (ReACTR also reduces the number of time-outs over Default in these cases but not to the same extent). The fact that SMAC reduces the number of runs which time-out is not surprising as it is tasked with reducing the PAR10 score which penalises failed runs more heavily. ReACTR on the other hand does not account for problem difficulty or failure and as such tends to reduce the median solving time. Secondly, these results consider the test set in isolation and assume there are no further instances to solve. By definition SMAC attempts to find a configuration which reduces the solving time of the entire runtime distribution of instances while ReACTR strives to adapt and find continually improving configurations over a potentially infinite stream of instances. In both cases increasing the size of the test set would likely skew the total solving time results in favour of the configurators as there is a greater time saving per instance when using a configurator. It is also worth remembering in this scenario that increasing the size of the test set allows ReACTR perform additional configuration while the configuration discovered by SMAC remains unchanged throughout.

Delving into the configuration performance of individual SAT domains we see that ReACTR's performance on the crafted track (Figure 4.11) was poorer relative to that on the random track (Figure 4.12). Figure 4.11a shows the cumulative average solving time of ReACTR only marginally improves upon the default configuration while SMAC achieves a modest, but much larger speed up (10.5% vs. 37.4%). Similarly, we see in Figure 4.11b that although the improvement through configuration is much greater on the N-Rooks benchmark, ReACTR's speed-up of 70% is poor relative to the 92% improvement achieved by SMAC.

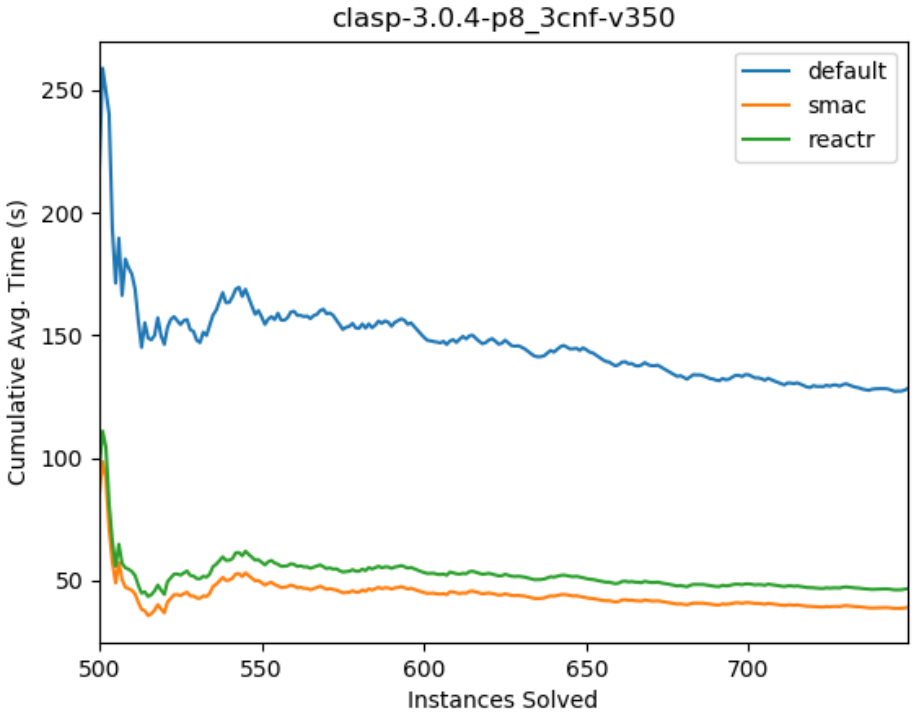


(a) Clasp Configuration: Graph Isomorphism dataset.

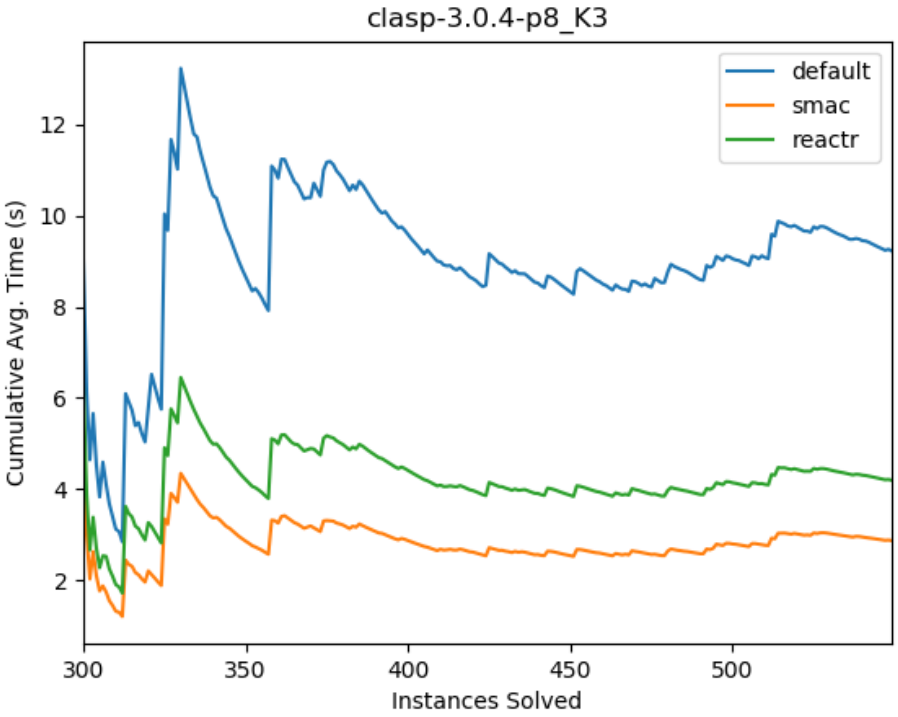


(b) Clasp Configuration: N-Rooks dataset.

Figure 4.11: ReACTR cumulative avg. solving times on Crafted SAT+UNSAT datasets.



(a) Clasp Configuration: 3CNF dataset.



(b) Clasp Configuration: K3 dataset.

Figure 4.12: ReACTR cumulative avg. solving times on Random SAT+UNSAT datasets.

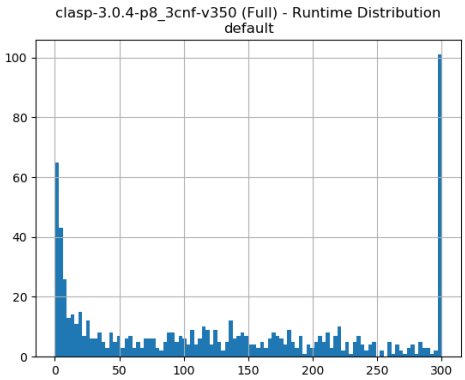
Turning our attention to the benchmarks from the random SAT+UNSAT track (Figure 4.12) we see that the situation for ReACTR improves slightly. Figure 4.12a that both configurators provide a large increase in solving speed over default and are approximately on a par with one another: 63.7% vs. 69.7% for ReACTR and SMAC respectively. Likewise Figure 4.12b shows that both configurators dramatically reduce the average solving time per instance albeit with a larger disparity in the percentage of improvement between them, 54.5% reduction for ReACTR as opposed to 68.8% reduction by SMAC.

Finally, the dataset 5SAT500 from the random SAT track exhibits the largest reduction in solving time through configuration. Table 4.5 shows that both configurators achieve an solving time improvement of over 99% on the dataset. While ProbSAT was unable to solve a single instance using its default settings all instances were solved in only a few minutes with configuration showing just how important quality configuration can be.

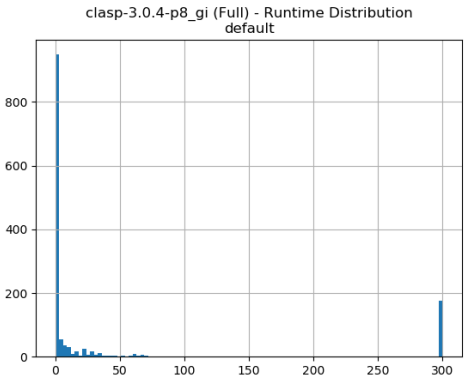
These results are positive and show that ReACTR is working as intended; configuring a solver while processing a stream of instances without any distinct training period. However, they are not in line with what we might have expected based on the results of the first set of experiments where ReACTR performed on a par with or exceeded the performance of SMAC on all datasets.

Investigating this further leads us to a few possible explanations. It is possible that ReACTR did not process enough instances in the warm-up (training) period to achieve good performance. This is unlikely as the dataset which exhibits the worst performance, Graph Isomorphism, solves the most training instances and has the longest warm-up period (see Table 4.3). Another possibility we consider is that ReACTR outperforms SMAC on larger configuration spaces. This holds true in the case of Lingeling however it is improbable as both CPLEX (74 parameters) and Clasp (75 parameters) have similarly sized configuration spaces to search.

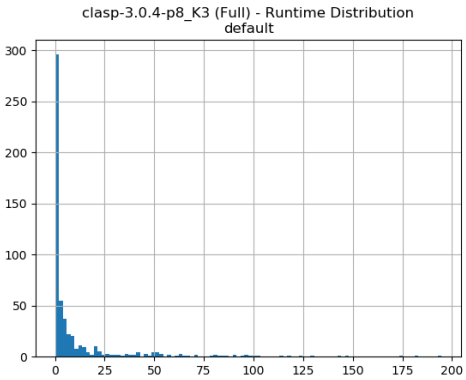
The most likely explanation for ReACTR's under performance is the instance runtime distribution of the new datasets. Figure 4.13 shows the distribution of default configuration run-times to have a heavy positive skew with a very light tail in most cases i.e. many extremely easy instances with few challenging instances. The case is further strengthened by observing that the two datasets exhibiting the strongest performance, 3CNF and Circuit Fuzz, also have runtime distributions with the heaviest tails (see Figures 4.13a and 4.13f) while ReACTR's weakest performance occurs on the dataset with the highest frequency of very easy instances, GI (Figure 4.13b). To quantify this we calculate ReACTR's median solving time for all datasets. The datasets where ReACTR performs well, arbitrary auctions, regions auctions, circuit fuzzing, and 3CNF have median solving times of 32.51, 31.13, 18.09, and 43.82 seconds respectively while



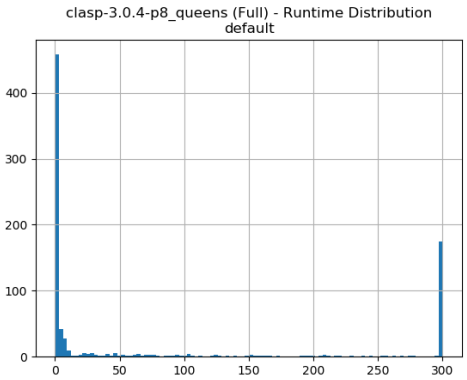
(a) Clasp: 3CNF



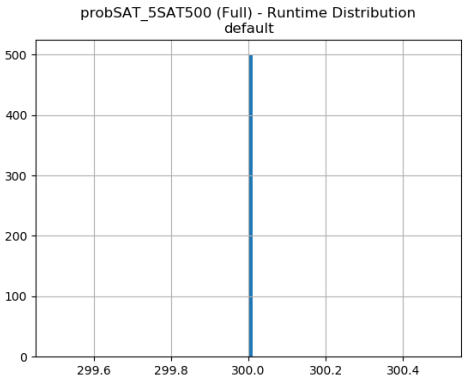
(b) Clasp: GI



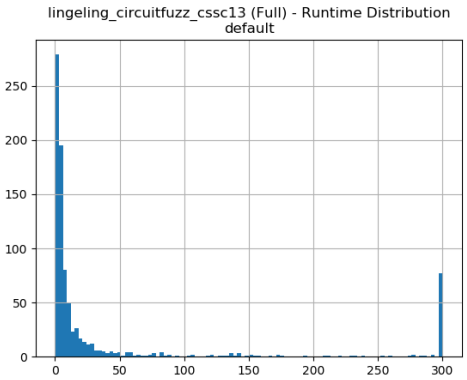
(c) Clasp: K3



(d) Clasp: N-Rooks



(e) ProbSAT: 5SAT500



(f) Lingeling: Circuit Fuzzing

Figure 4.13: Default configuration solving time distributions for all benchmarks.

the median times for the benchmarks where ReACTR under performs, GI, N-Rooks, K3, and 5SAT500 are 0.12, 0.75, 0.43, and 0.85 seconds. Clearly from this evidence we see that an abundance of instances that can be solved extremely quickly negatively impacts on ReACTR's configuration procedure.

While this explains the conditions which lead to ReACTR's poor performance, it does not explain why SMAC outperforms ReACTR in these situations. We believe multiple factors are at play here. Firstly, as ReACTR does not weight instances by solving time and instead relies on ranking it is likely that small fluctuations (for example due to system scheduling) will have a greater impact on this ranking and derail the search procedure. This lack of weighting is a limitation of the ReACTR system (explained in detail in Section 6.2.5) that means that ReACTR may over fit the more common easy instances at the expense of solving performance on the more difficult instances which have a larger impact on the overall solving time. Secondly, as we have shown previously, SMAC's performance can be improved when its upfront training is performed on easier instances. We saw this effect in Figures 4.3 and 4.5 where training on a dataset with a lower number of goods or bids can outperform SMAC trained on a randomly selected sample of instances. This phenomenon has also been noted independently and proposed as a method of improving configuration performance on more difficult instances [SH13].

4.4 Chapter Summary

In this chapter we presented two concrete instantiations of the ReACT framework, ReACT and ReACTR. We motivated and discussed the implementation choices of the various component parts and empirically evaluate both configurators against the state-of-the-art offline configurators on a variety of benchmarks with favourable results.

Chapter 5

Leaderboard, Candidate Selection and Instance Ordering

Summary. *In this chapter we outline the intricacies of the various ranking mechanisms which are applicable to the ReACT framework. We discuss each of their relative strengths and weaknesses, in particular we justify the use of the Bayesian ranking system TrueSkill as the cornerstone for our primary ReACT framework instantiation, ReACTR.*

With an understanding of the leaderboard and ranking mechanism in place we then describe how these methods and others are applied to provide a robust method for candidate selection. Here we assess the performance achievable by different selection metrics and methods of combining these metrics. Additionally, we demonstrate that the optimal combination depends on the composition of the incoming instance stream and that there is no fixed hierarchy to these blending methods.

The contents of this chapter has appeared in the peer-reviewed proceedings of ACM Symposium on Applied Computing 2016 [FO17] and the journal Fundamenta Informaticae [FO19].

5.1 Leaderboard and Ranking

The previous chapter demonstrated that the leaderboard and ranking system is at the core of the ReACT system. The leaderboard is utilised by a number of components in the framework: the selection procedure to select quality configurations to run; the

removal component to identify under performing configurations to purge; and even by the generation method to single out promising configurations to be used as a template for new configurations.

Naturally, because of the important function that ranking plays in the framework we look for a ranking method that is both easy to compute and accurate. For this reason we turn our attention to the world of competitive games where there is a large body of research on the ranking of players and teams in multiplayer games such as chess and online video games [HMG06, KGK16, TLZ+05, Gli98]. Being able to accurately gauge a global ranking based on head-to-head competitions is very valuable for handicapping, qualification invitations and fair match making. In this remainder of this section we outline a number of competing ranking systems, the concepts behind them, as well as their relative strengths and weaknesses.

5.1.1 Bradley-Terry Model

Developing a method to produce a ranking given a series of pairwise comparisons has been an area of study for many years. One of the seminal papers in the field is by R. A. Bradley and M. E. Terry who proposed the so called Bradley-Terry model in 1952 [BT52], though it was studied much earlier by Ernst Zermelo[Zer29]. This model provides a probability estimate of one player beating another given a set of paired comparisons. More formally this can be expressed as

$$\mathsf{P}(i > j) = \frac{\gamma_i}{\gamma_i + \gamma_j}$$

where γ_i and γ_j are positive scores associated with the performance of players i and j respectively. This could, for example, be derived from the number of times i wins over j in a football season. $\mathsf{P}(i > j)$ gives the probability that i wins or is preferable to j. The parameters of this model are fitted by means of maximum likelihood estimation [H+04]. The log-likelihood is given by

$$\ell(\boldsymbol{\gamma}) = \sum_{i=1}^{m}\sum_{j=1}^{m}[w_{ij}\ln\gamma_i - w_{ij}\ln(\gamma_i + \gamma_j)]$$

where m is the number of competitors and w_{ij} is the number of times competitor i beats competitor j.

This model is very useful having many practical applications such as evaluating results from sensory panels [Bra53] and ranking results for search engines [RJ07]. The model

also lays the foundations for much of the work that followed such as Elo, Glicko, and TrueSkill rating systems which we will discuss in the upcoming sections.

5.1.2 Elo

One of the most widely known ranking systems is the Elo rating system which was developed by Arpad Elo in the late 1950s and adopted by the International Chess Federation (FIDE) in 1970 [Gli95]. Elo-type systems are now widely used for many different games (albeit parameterised differently) from soccer to scrabble.

The Elo rating system asserts that a players skill can be modelled by a normal distribution (though many chess rating schemes now use a logistic distribution as it provides a better fit). Furthermore, Elo puts forward the idea that the points difference between two players' ratings is a probability predictor for the match outcome. In chess an average player is expected to have a rating of about 1500 points (although this varies between federations as there is no single implementations) [GJ99]. Somewhat arbitrarily, a 200 point difference in skill equates to an approximately 76% chance of the higher rated player winning. The formula for computing win probability is

$$E_A = \frac{1}{1 + 10^{(R_B - R_A)/400}}$$

where R_A and R_B are player A and B's respective ratings. E_A is player A's expected score (probability of winning plus half the probability of a tie).

Example 5.1.1. Consider the case where player A has a rating of 1700 and player B has a rating of 1500. We compute the probability of player A winning as:

$$E_A = \frac{1}{1 + 10^{(1500-1700)/400}} = 0.76$$

The rating points assigned by the Elo system during an update are zero-sum; if player A gains points in a score update player B loses the same number of points. The number of points wagered is not fixed and is instead dependant on the expected outcome of the game and the K factor. The more unexpected an outcome the more points are lost by the party which has been upset. This is logical as a player with a much lower expected score triumphing over a player with higher expected score suggests that the assigned scores are incorrect and a large update is required. The K factor is a type of weighting that is applied to updates. Generally, newer players are assigned a higher K factor by the organising body (e.g. FIDE) that drops as the player gains experience. This scheme

allows new players to quickly arrive at their new score while allowing more experienced players to maintain a relatively stable score.

Formally the updates to a players score are performed as follows:

$$R'_A = R_A + K(S_A - E_A)$$

Here, R'_A is the updated score, R_A represents the previous score, K is the applied K factor, S_A is the score achieved in the game, and E_A is the players expected score for the game.

Example 5.1.2. To continue the previous example, player A's rating following a win over player B would be updated in the following way (assuming the FIDE K value of 20):

$$R'_A = 1700 + 20(1 - 0.76) = 1704.8$$

The Elo rating system is conceptually easy to understand and computationally inexpensive to run. This, combined with the strong approximation of player skill that Elo provides, has allowed it to become the defacto standard for ranking algorithms. Despite this, the Elo system is not without its flaws. Primary amongst these, particularly in the context of a ranking system used in ReACT, is the fact that it is only designed for two player games. To calculate ratings for more than two players, such as the races at the core of the ReACT framework, it is necessary to perform a pairwise comparison for all pairs of competitors in the race. When adopting such an approach one must be careful with the sequencing of the comparisons and score updates so as to reflect that the races occured simultaneously.

Another issue facing Elo is its inability to model draws. Within the Elo system draws are considered a half win and half loss. This overlooks the fact that draws can supply valuable information. For example if a new player with a low Elo score draws with a player holding a much higher score we can infer that the players are somewhat equal in skill level. This also leads to difficulty interpreting the expected score of games. Remember that expected score is the probability of winning plus half the probability of a draw. Consider an expected score of 0.6, on one extreme this could suggest that the player has a 60% chance of winning and 0% chance of a draw, on the other it could imply a 20% win chance and 80% draw percentage, or anything in between.

The final issue worth mentioning is that Elo fails to provide confidence in the rating it has assigned. A player that has a score of 2000 after 500 games is more likely to be accurate than someone who has the same score after just 10 games, yet this is not

explicitly modelled. Careful selection of the K factor can help alleviate some of the issues surrounding score uncertainty, however this is an art rather than a science. The drawback is normally remedied by considering a player's rating as being provisional until they have completed a certain number of games, typically twenty or thirty. This is a deal breaker for using Elo in ReACT-based systems as we aim to assess a configurations confidence as quickly as we possibly can.

5.1.3 Glicko and Glicko-2

The Glicko rating system, introduced in 1999, aimed to address some of the perceived short comings in the Elo system [Gli99, Gli98]. Primary amongst the improvements introduced by the Glicko system is the inclusion of a so called rating deviation, RD, for each player. Adopting this methodology allows Glicko to do away with the provisional ranking period used in Elo and to more accurately describe the system's confidence in a rating at any given time. The rating deviation allows a player's skill to be described in terms of a confidence interval rating than a single rating as is the case in the Elo system. In Glicko a player's rating is provided as a 95% confidence interval bounded by their score minus 1.96 times their rating deviation and their score plus 1.96 times their rating deviation respectively. To give a concrete example, consider two players, P_a with score of 1700 and a RD of 30, and P_b with the same score, 1700, but a much higher RD of 120. This gives us a 95% confidence interval for both players of:

$$P\_a = (1700 - 1.96 \times 30, 1700 + 1.96 \times 30) = (1641, 1759)$$

$$P\_b = (1700 - 1.96 \times 120, 1700 + 1.96 \times 120) = (1465, 1935)$$

Under the Elo system both of these would be considered equivalent, however by adding the rating deviation it is apparent that the system is far more confident in P_a's skill, this shown by the narrower skill range. On the other hand, the wide rating interval of P_b shows that the system is not yet confident in the players skill, believing it to range somewhere between 1465 and 1935.

Unlike the Elo system, ratings updates in Glicko are not zero sum. The number of points gained or lost takes into account the systems confidence in the rating of each player. The system will not drastically change a the score for a player with an established rating that it is confident in. On the other hand if a new player (or a player the system is not confident in) scores a suprising win against an established higher ranked player the system is likely to make a large update to their score. Intuitively this makes sense as triumphing over a stronger opponent suggests that their current score is too low. The

specifics of these updates are beyond the scope of this theses, but we refer the reader to the Glicko paper for full information on the algorithms [Gli99].

Glicko-2, introduced in 2001, builds on the foundations laid by the original Glicko system by introducing a measure of performance volatility in addition to the rating and rating deviation measure [Gli01, Gli13]. This extension models the variability of a players performance within a given "rating period". A rating period consists of a number of games which are grouped together and considered to occur simultaneously for the purpose of ranking updates. A player who performs consistently during games in the rating period will have a low volatility score, erratic performances produce a larger volatility rating. We mention this extension primarily for completeness as the ReACT framework does not derive much benefit from it. This is due to the fact that ReACT must use a short rating period of a single race which makes it impossible to detect volatility in performance. It is necessary to do this in order to produce ratings as quickly as possible, without ncurring the overhead of waiting for a number of races to complete.

Though Glicko addresses some of the short comings of the Elo rating system it still lacks the ability to rate competitions with more than two players. This is a requirement for the ReACT system as races will almost always consist of more than two configurations. In order to use Glicko for games involving three or more players, all combinations of pairs must be created and updated independently in order for either of these classic approaches to work.

5.1.4 TrueSkill

To solve the multi-player problem for online games, the ranking algorithm TrueSkill was introduced in 2006 [HMG06]. TrueSkill is able to rank multiple players (and teams of players) as well as explicitly modelling draws. TrueSkill draws much inspiration from the Glicko system in the way it models player skill; the system tracks both a player's average skill, μ, as well as the degree of confidence in that score (standard deviation), σ, assuming a Gaussian distribution for a player's skill. TrueSkill uses Bayesian inference to perform updates on player's rankings [HMG06]. Each player's skill rating and deviation are used as the prior. After a tournament, all competitors are ranked based on the result with surprising results inducing a larger update than expected results. The result of this is that a player that is expected to win (higher μ value) gains little by beating a lower ranked opponent. However, if a lower ranked player beats a higher ranked player, then the lower ranked player will usually receive a large increase in their μ value (depending on the systems certainty). As a player competes in more

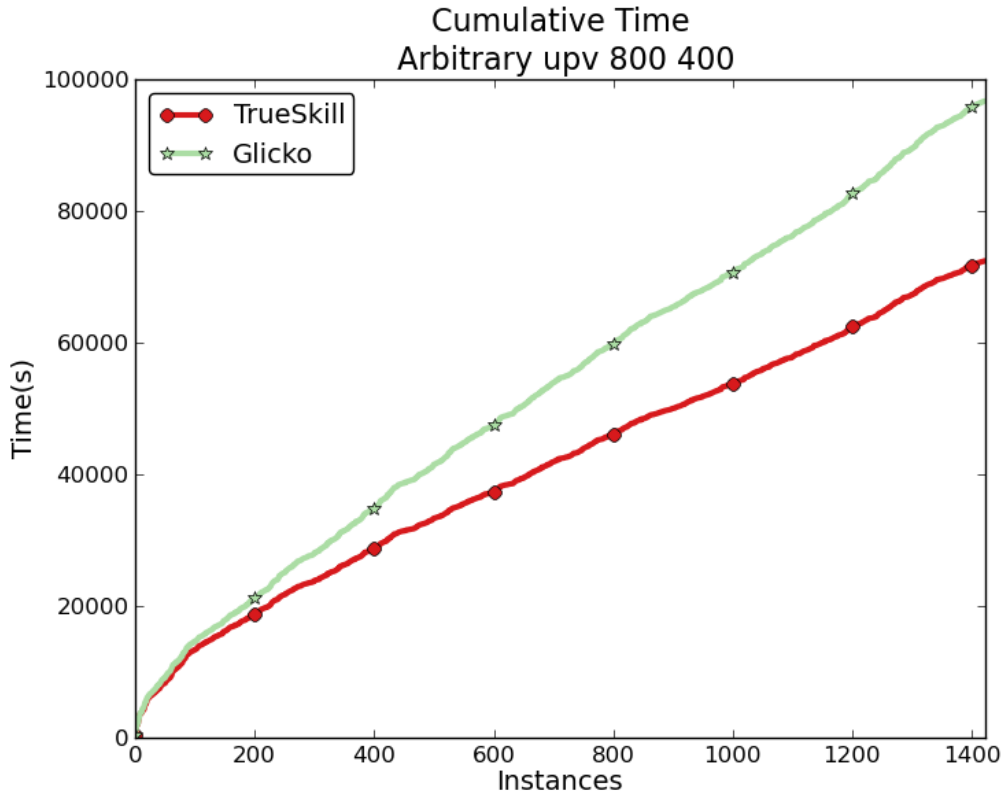


Figure 5.1: Cumulative solving time using Glicko and TrueSkill for ranking.

tournaments TrueSkill becomes more confident in the μ that is assigned and so the uncertainty value, σ is reduced after every tournament played.

TrueSkill is quite efficient when performing updates due to the use of approximate message passing on a factor graph representation of the probabilistic model. Factor graphs are a probabilistic graphical modelling technique used to encode probabilistic distributions. The graph is a bipartite graph consisting of two types of nodes, variables which represent a known or sought value, and factors which define the relationship between variables in the graph. Approximiate message passing (an efficient iterative approach for solving the standard linear regression problem) is used to compute the marginal distribution over the variables [DMM09]. The specifics of this algorithm are beyond the scope of this description, however, we refer the reader to the TrueSkill paper for full details [HMG06].

5.1.4.1 Performance of ReACTR using TrueSkill Ranking vs. Glicko Ranking

For ReACT, the confidence metric provided by Glicko-2 and TrueSkill is a highly desirable feature that could help determine whether it is worth continuing to evaluate a configuration or whether it can be safely discarded. We implemented versions of

the ReACTR algorithm utilising both ranking algorithms. Figure 5.1 shows the result of one of these experiments on the set of arbitrary combinatorial auction instances modeled as mixed integer programming problems and solved using the CPLEX solver (Section 4.3.4 provides a more detailed description of this setup). This figure shows the cumulative solving time for solver tuned with ReACTR algorithm and using the two ranking methods. We see that TrueSkill is able to help bring better configurations to the top, resulting in superior overall performance in the long run.

5.2 Candidate Selection

ReACT's parallel racing mechanism provides it with a way to evaluate multiple configurations side by side. This is what allows ReACT to function in an online fashion. An important part of this parallel racing mechanism is being able to correctly identify which configurations should be evaluated on each instance. Parallel algorithm portfolios face a similar challenge when selecting which solvers to run on each instance. Portfolios generally aim to maximise coverage of the number of instances by running complimentary solvers [HKMO14a]. This is often achieved by looking at instance features and predicting which solvers will work well on which instances [XHHLB08]. ReACTR does not use instance features, but instead relies on past performance data and a ranking system to determine which instances to select for each instance. As such, it performs in a per-set manner rather than on a per-instance basis.

Given that only a limited number of configurations may be run in parallel, the challenge lies in balancing the exploration of the large configuration space with maintaining good performance by using "proven" configurations. This is known as the exploration versus exploitation trade-off and is a well-studied issue [VM05, KP14].

The candidate selection procedure plays a vital role in the ReACTR system and its overall performance. In this section we explore various combinations of performance metrics in order to identify those which provide the best performance.

5.2.1 Selection Metrics

One of the aims of this chapter is to more fully understand the candidate selection procedure used in ReACTR. ReACTR's current candidate selection policy uses TrueSkill as its performance metric. In our experiments we evaluate a number of different performance metrics, and various methods of combining them. The Epsilon-Greedy approach is still used (as outlined in Section 4.3.2), so a proportion of candidates selected are always random to ensure a certain level of exploration within the parameter pool. In addition

to these, a number of "good" candidates are selected based on various performance metrics as discussed below.

In addition to testing TrueSkill as a performance metric we also look at a number of simple metrics based on the number of wins and how recent those wins were. In this context a winner is defined as the configuration that outperformed its competitors by solving a problem instance first. In our experiments we use a shorthand to label the mix of selection procedures used; this takes the form nSEL where n is the number of configurations selected by the selection method SEL. The selection method abbreviations and explanations are as follows:

- TS (TrueSkill) - These configurations are selected according to their TrueSkill score.
- LW (Last Winners) - Keeps a list of the most recent race winners. It then selects the n most recent unique configurations. e.g. 1LW uses the configuration which solved the previous instance fastest. 2LW uses the winners of the races on the two most recent instances (or potentially more if both were solved by the same configuration) etc.
- WIN# (Win Count) - The number of times a certain configuration has performed best. It is important to note this is not normalised based on the number of runs the configuration has taken part in. e.g. if a configuration has run in ten races and won five, its win count is 5. WIN# sorts the configurations by descending win count and takes the n configurations with the most wins.
- WIN% (Win Percentage) - Similar to Win Count but normalises for the number of runs in which a configuration has participated. This normalisation makes Win Percentage a better metric as it is no longer biased towards configurations with more runs. e.g. if a configuration has run in ten races and won five, its win percentage will be 0.5. Again WIN% sorts the configurations by descending win percentage and takes the n configurations with the highest win percentage.
- DEF (Defeats) - Defeats must be used in conjunction with another performance metric. Defeats chooses the n configurations that have defeated the configuration selected by that metric the most. e.g. Given the mix of selectors 1WIN#1DEF, 1WIN# will select the configuration with the highest number of wins, $conf_w$. 1DEF will then order the remaining configurations by the number of times they have competed against $conf_w$ and won.
- RAND (Random) - Selects configurations uniformly at random.

We typically use multiple selection procedures that are applied in the order they are listed. For example 2TS2LW2RAND will select two configurations using TrueSkill ranking(TS) first, then two using the Last Winners procedure (LW) and finally two at random (RAND). If a configuration has already been selected by a previous selection procedure then the next matching configuration which has not already been chosen. If a no configurations match a selection criteria the configurations are instead selected at random.

A number of baseline selection strategies are also provided. Due to the differing nature of the runs, different baselines are proposed for static and dynamic datasets. In the static case, three baseline strategies are provided. First, the *Oracle* baseline shows the results for the virtual best system, which selects the best solver for every instance. This is the best performance that is possible to achieve. *Single Best* shows the best single solver that minimises the overall total solving time. *Random* selects the required number of solvers at random for each instance and chooses the best solving time from these. In the dynamic case the baseline is the average running time of the untuned solver.

5.2.2 Datasets and Instance Generation

For these experiments we use three different datasets. Two datasets, PROTEUS-2014 and SAT12-ALL, are taken from the Algorithm Selection Library (ASLib) [HKMO14a, BBD+12, BKK+16]. These are static datasets with a fixed number of solvers where run times for all solvers are precomputed. In this scenario each solver in the dataset equates to a different configuration. This allows us simulate ReACTR runs by iterating over the instances in these static datasets in the same way that ReACTR would but using the precomputed solving times instead of performing a solver run with a configuration. As the pool of "configurations" (solvers) is fixed we do not employ the removal and replacement procedures normally used in ReACTR.

Excluding ReACTR's removal and replacement procedure also allows us to study the effects of ordering and candidate selection in isolation. The caveat to this is that the interaction between these procedures cannot be studied and as such full ReACTR runs may behave differently. However, it is necessary to use static datasets as conducting a full run using the ReACTR system requires a large number of CPU hours (or days) and does not readily parallelise. As the simulated run is quick to compute, the static results are based on 100 runs that allows us to better estimate the variability in performance of the selection methods and compare them more accurately.

The first static dataset, PROTEUS-2014, comprises 4021 constraint satisfaction prob-

lems solved using 22 different solvers. We use Mistral as the default solver [Heb08]. This is in keeping with the methodology used in the paper that created this dataset [HKMO14a]. Instances that the default solver solved in less than two seconds were filtered out. Additionally any instances where all solvers hit the 3600 second cut-off time were removed. This left 2595 instances for analysis which were neither too easy nor too difficult. Four different orderings of these 2595 instances were then considered: *Lexicographic*, *Shuffled*, *Easy-to-Hard*, and *Hard-to-Easy*. *Lexicographic* orders the instances based on the lexicographical ordering of the instance file paths. This is important because the way the dataset was created means similar instances tend to be clustered together in the same folders e.g. 8-Queens would be next to 9-Queens in the n-Queens folder. *Shuffled* randomly orders the set of instances. *Easy-to-hard* and *Hard-to-Easy* sort the instances from fastest solving time to slowest, and vice-versa, using the default solver, Mistral.

For our feature ordering experiments we used a subset of the PROTEUS-2014 dataset comprising 623 instances. This subset was chosen so that all feature values were present and did not need to be computed. All 198 instance features given as part of the ASLib dataset are used; these are described in more detail in Section 5.2.3. These feature values are used to sort the instances in both ascending and descending order.

The second static dataset, SAT12-ALL, contains a mix of 1614 Boolean Satisfiability (SAT) instances taken from SAT competitions. These are solved using 31 different solvers. Lingeling is selected as the default solver for this dataset [Bie10]. Lingeling is a highly parameterised SAT solver which has performed well in recent configurable SAT solver challenges[HLB$^+$17]. Again, any instances that take less than 2 seconds to solve with the default solver, Lingeling, or are unsolved by any solver within the 1200 second cutoff time, were removed. The remaining 1474 instances were again ordered in the four orderings outlined above. Similar to PROTEUS-2014, for our feature ordering experiments we only used a subset of the instances where all instance feature values are available. This gave a dataset comprising 721 instances. We ordered by all 115 instance features, which will be described in Section 5.2.3, in both ascending and descending order.

In addition to the two static datasets, a dataset consisting of combinatorial auction MIP instances was also used. This benchmark does not have pre-calculated run times and must be solved by the CPLEX solver (being configured by the ReACTR system) to determine the runtime for each instance. Due to computation time involved in running ReACTR over a large number of instances (in the order of a day for each configuration run), each run using this dataset is evaluated only 10 times. All dynamic experiments

were run on a system with 2×Intel Xeon E5430 processors (2.66Ghz) and 12 GB RAM. The system has 8 available cores but only 6 are used so as to allow room for background processes and other overhead without affecting timing. The Algorithm Configuration Library(ACLib) [HLF+14] framework was used to run the dynamic experiments.

Four different combinatorial auction domains are combined to create the dynamic benchmark based on combinatorial auctions. These instances are generated using the Combinatorial Auctions Test Suite (CATS)[1]. The four domains generated are arbitrary, paths, regions, and scheduling. Arbitrary combinatorial auctions have no clear connection between the goods being auctioned e.g. antiques. The Paths domain models combinatorial auctions based on paths in space, generally a route between two points e.g. truck routes. Regions simulates combinatorial auctions where the adjacency in space of goods matters e.g. parcels of land. Scheduling instances simulate auctions where temporal scheduling auctions e.g. selling time on a machine. Arbitrary and regions instances were both generated using 100-100 goods and 100-2000 bids, paths instances have between 512 and 2048 goods with 3000-10000 bids, while scheduling instances have 128-256 goods and 3000-10000 bids. These parameters were chosen in order to create instances that proved challenging for the mixed integer programming (MIP) solver used, CPLEX [IBM14]. CPLEX 12.6 is used with 74 discretised parameters as provided by ACLib. The first 200 instances for each domain, that were solvable within 30 to 600 seconds using the default CPLEX configuration, are then merged to create a benchmark with 800 challenging but solvable instances. When solving using ReACTR a cut off time of 180 seconds was used, which allows room for the configurator to improve over the default solver configuration.

5.2.3 Features

The SAT12-All dataset uses 115 SAT instance features that are used by SatZilla in the 2009 SAT competition [XHHLB09]. For the sake of brevity we will not exhaustively list all features, however, they can be divided into subgroups: problem size features, variable-clause graph features, variable graph features, balance features, proximity to Horn formula, DPLL probing features, local search probing features, survey propagation features and clause learning features. A technical report describing these features in more detail is available [XHHLB12].

The PROTEUS-2014 dataset contains instances and features used in the Proteus hierarchical portfolio of solvers [HKMO14c]. Proteus uses both SAT and CSP solvers, sometimes encoding a CSP problem in SAT and solving using a SAT solver if this is

[1] `http://www.cs.ubc.ca/~kevinlb/CATS/` [LBPS00c]

expected to improve performance. Therefore, the PROTEUS-2014 dataset contains a mixture of both CSP and SAT features. There are 36 CSP features which were originally used in CPHydra [OHH+08b]. These include statistics about the domain sizes, the type of constraints and the progress of the Mistral solver when run for 2 seconds. The SAT features used for the PROTEUS-2014 dataset are similar to those of SAT12-All outlined above. The one notable difference is that the features are calculated from multiple different SAT encodings of the CSP instances (support, direct order and direct).

5.3 Experiments on Fixed-set Solver Datasets

5.3.1 Lexicographical and Runtime-based Ordering

These experiments have two goals, to explore which candidate selection strategies work best, and also what effect the instance ordering has on the overall solving time. To do this different performance metrics (described in Section 5.2.1) are used and combined in a number of ways and then evaluated on multiple instance orderings.

Figure 5.2 shows box plots representing the solving time using various selection mechanisms on the SAT12-ALL dataset which has been ordered lexicographically. The sampling methods are ordered by their median values, which is indicated in the boxplot by the red central line. The blue box shows the first and third quartiles (the 25th and 75th percentiles). The lower whisker shows $Q1 - 1.5 \times IQR$, while the upper whisker shows $Q3 + 1.5 \times IQR$, where $Q1$ and $Q3$ are the first quartile and third quartile, respectively, and the inter-quartile range, IQR is the difference between them. Outliers are marked with black crosses.

Three baselines are included to give some context to the results. $Oracle$ chooses the best possible solver for each instance. $Random$ selects six solvers at random for each instance and logs the best time achieved from the six. $SingleBest$ is the solver which has the lowest overall solving time over all instances. All of the selection methods shown outperform both the $Random$ and $SingleBest$ baselines which shows that using any of the ReACTR selection methods alone even without the ability to modify or introduce new solvers or configurations still gives an improvement and is a worthwhile endeavour. Though not shown, this result holds across all orderings of both PROTEUS-2014 and SAT12-ALL datasets.

Figure 5.2 shows a jump in solving time when transitioning from 1LW 5RAND to 5WIN% 1RAND. All selection policies to the left of 1LW 5RAND in Figure 5.2 use Last Winners (LW) as a selection component, after that point only combinations of

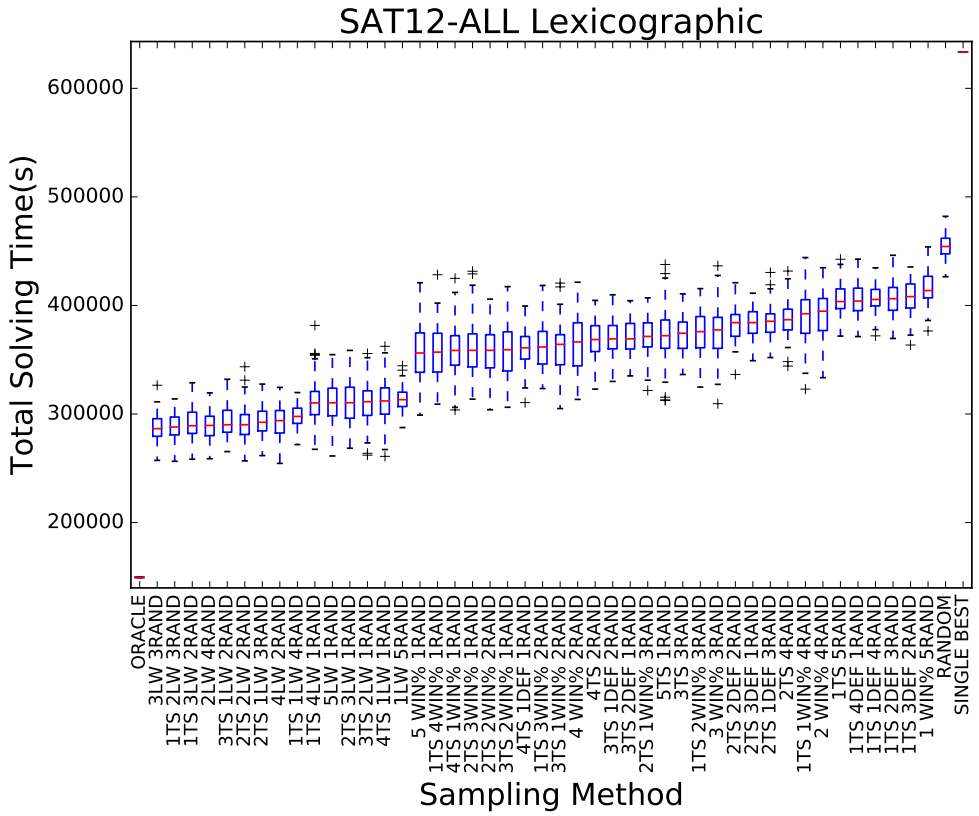


Figure 5.2: Comparison of all candidate selection methods on the SAT12-ALL dataset ordered lexicographically (Avg. of 100 runs).

TrueSkill, Win Percentage and Defeats (described in Section 5.2.1) are used instead. It is clear that having Last Winners as part of the selection policy is important for SAT instances that are lexicographically ordered. When the instances are ordered lexicographically rapid changes in the domain type occur, for example when switching from a folder containing hardware verification to cryptography instances. By using the last winning configuration these changes are detected quickly rather than waiting for a TrueSkill score or Win Percentage to rise sufficiently for the candidate to be selected by those metrics. Supporting this hypothesis is that none of the other SAT orderings, which are shuffled instance-wise, show this sharp jump.

There is a smaller jump also visible between the ninth and tenth box plots (1TS 1LW 4RAND and 5LW 1RAND) which appears to be caused by the reduction in the number of random candidates. Last Winners, in the SAT lexicographical case, seems to need a larger number of random candidates included. Since the domains encountered are changing rapidly, a heavy emphasis on exploration within the leaderboard is required. Random selection provides this exploration and allows the selection policy to quickly discover the best solver for the current instance type, while using Last Winners allows

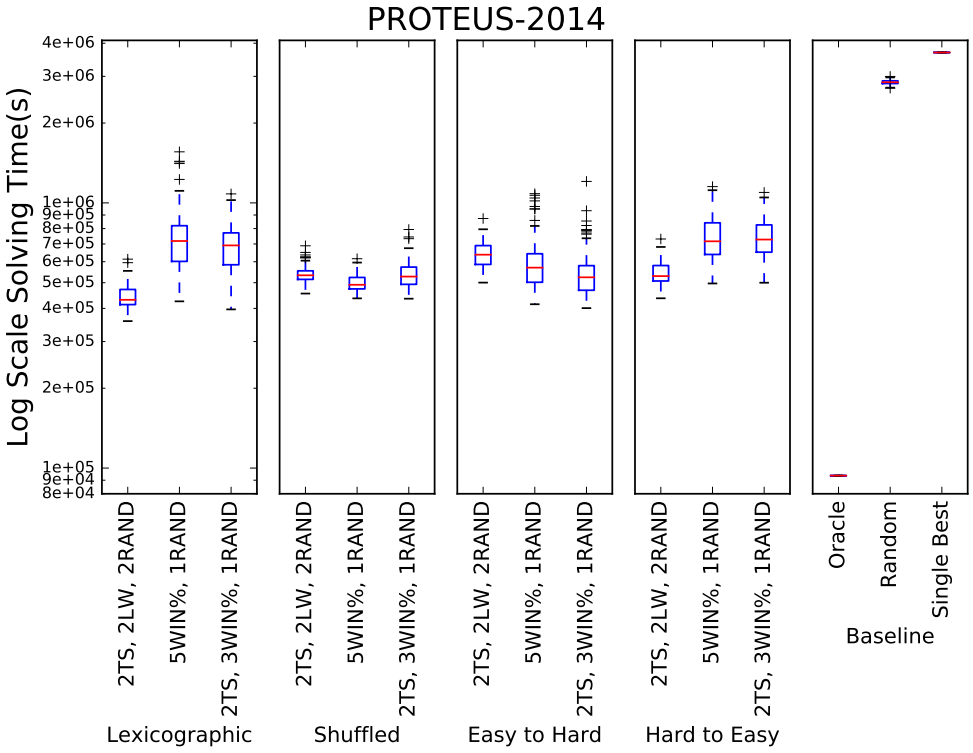


Figure 5.3: The solving time for the best performing candidate selection mechanisms for each PROTEUS-2014 ordering when evaluated on all other orderings (Avg. over 100 executions).

the best solver to be kept and used for upcoming instances. This behaviour is not typical for the other orderings or datasets where normally more exploitation using only one or two Random exploration candidates is favoured.

Note that selection policies that include Defeats appear to under-perform. When examined more closely Defeats adds little improvement over selecting randomly. In general, it appears the majority of any good performance observed when Defeats is used can be attributed to the TrueSkill part of its composition.

Figures 5.3 and 5.4 show the best selection policies from each individual ordering evaluated on all of the other instance orderings for both PROTEUS-2014 and SAT12-ALL respectively. The y-axis is shown in a log scale in order to include the baseline results. Again we see that all selection policies outperform the Random and Single Best baselines. In both cases we see that instances that are ordered lexicographically are solved faster than all other orderings using the optimal selection policy. *Shuffled* instances, though not solved fastest, have the smallest deviation in solving time. This is most obvious in the case of Proteus *Shuffled*.

The lexicographically sorted Proteus dataset is solved fastest using a combination of

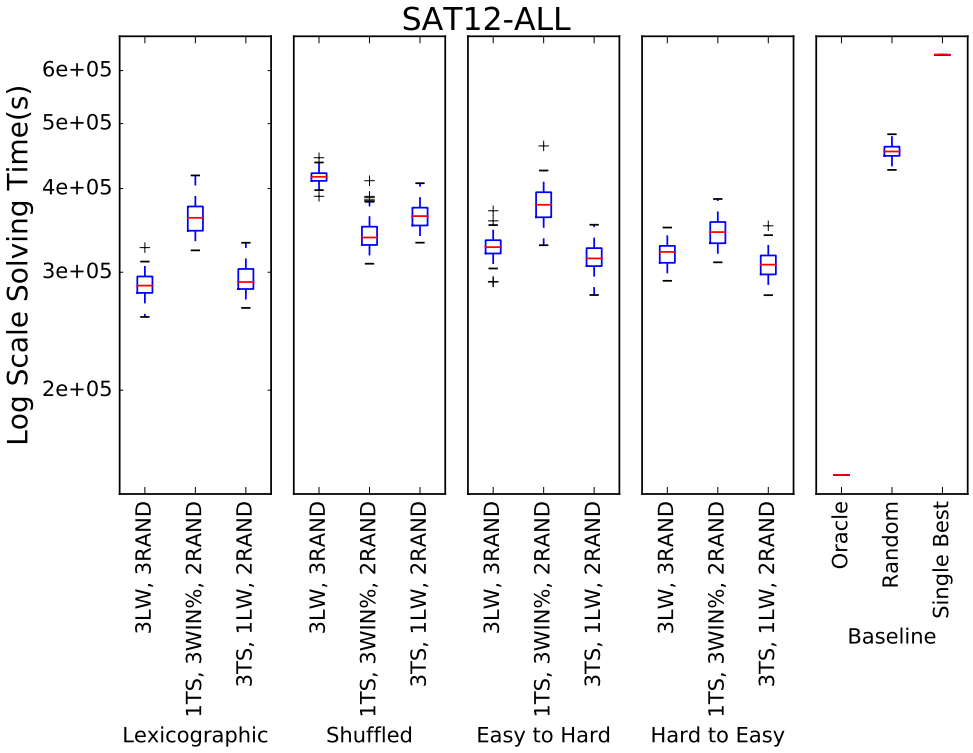


Figure 5.4: The solving time for the best performing candidate selection mechanisms for each SAT12-ALL ordering when evaluated on all other orderings (Avg. over 100 executions).

TrueSkill and Last Winners. This is in keeping with what was shown in Figure 5.2 and for possibly the same reasons outlined previously. The other orderings appear to express a preference for certain candidate selection models as well regardless of dataset. Shuffled instances from both Proteus and SAT are are solved most quickly using a Win Percentage-based system (5WIN% 1RAND and 1TS 3WIN% 2RAND, respectively). Similarly, selection policies using TrueSkill and Last Winners perform best on instances that are ordered from hard-to-easy (2TS 2LW 2RAND for Proteus and 3TS 1LW 2RAND for SAT). The only ordering that bucks this trend is *Easy-to-Hard*, favouring a TrueSkill and Win Percentage-based model for Proteus (2TS 3WIN% 1RAND) but a TrueSkill and Last Winners approach for SAT (3TS 1LW 2RAND).

Some commonality amongst the most preferred selection policies is also visible. Figure 5.3 shows that Proteus *Lexicographical* and *Hard-to-Easy* fare best using 2TS 2LW 2RAND, while Figure 5.4 shows both SAT *Easy-to-Hard* and SAT *Hard-to-Easy* agree on 3TS 1LW 2RAND as the optimal selection policy. This may signify that these ratios are the right balance for those particular datasets. Unfortunately there is no general consensus on ratios across datasets, though Proteus *Hard-to-Easy* does have 3TS 1LW 2RAND as a second choice suggesting some commonality is present.

5.3.2 Feature-Based Ordering

Our initial ordering experiments in Section 5.3.1 showed that instance ordering by runtime does make a difference to the configuration process. Positive improvements in the total solving time can be achieved based on simple runtime or lexicographical ordering alone. In this section we extend these ordering experiments to investigate the effect of ordering instances based on their feature values. Instance features can reveal much about the instances being processed such as their size, structure, and how the instance changes when a solver is run on it for a short period of time; we refer to running a solver in this way as "probing". Most features can be computed quickly, often by just parsing the instance file. Understanding how ReACTR performs on orderings based on different instance features provides a greater understanding of the what impacts configurator performance and how this might be exploited for improvement. These experiments show where ReACTR achieves its best and worst performance in certain domains.

Getting these insights leads to a greater understanding of how ReACTR will perform in real world scenarios. Companies are often faced with a stream of increasingly large problems to solve. For example a ride-sharing company might see the stream of instances it must solve grow in size and complexity as the company expands over time, or there can be similar effects between off-peak and rush-hour periods at a daily level. Similarly a factory may face increasingly difficult scheduling problems as the company employs more staff or takes on more orders. Though there is no direct control over the type of instances encountered it is still important to be aware of how the configurator responds to the size of certain features increasing and decreasing so as to avoid any pitfalls.

The previous experiments focused on which of the selection policies is most preferred whereas the goal of these experiments is to study the effect of instance feature ordering. As such we use a single candidate selection policy for the experiment: 2TS 2LW 2RAND comprising two TrueSkill, two Last Winners and two Random selectors. This candidate selection policy was chosen as it performed best in two of the previous PROTEUS-2014 experiments. Candidate selection policies containing a mixture of TrueSkill and Last Winner also performed well on the SAT12-All dataset making this a good compromise choice for both datasets being investigated.

The instances in the SAT12-All dataset are each described using 115 features. These are ordered both ascending and descending to give a total of 230 different instance orderings. Each instance ordering is evaluated 100 times using the simulated ReACTR run methodology outlined in Section 5.2.2. Each run uses a different seed

so that any stochastic parts of the selection procedure produce different results. We compared the distribution of total runtimes for each instance ordering against the distribution given by 100 runs of both Random and Easy-to-Hard orderings using a statistical hypothesis test (Student's t-test). Of the 230 different SAT feature orderings 223 were statistically better (P=0.01) than Random ordering while 200 were statistically better than the Easy-to-Hard ordering. No ordering was statistically worse than Random, though four orderings did have worse average runtimes (`UNARY` and `POSNEG_RATIO_CLAUSE_max` sorted both ascending and descending). Eight orderings were statistically worse than the Easy-to-Hard ordering (`POSNEG_RATIO_CLAUSE_min`, `POSNEG_RATIO_VAR_min` (ascending and descending), `gsat_FirstLocalMinStep_CoeffVariance`, `POSNEG_RATIO_CLAUSE_max` (ascending and descending) and `UNARY` (ascending and descending)).

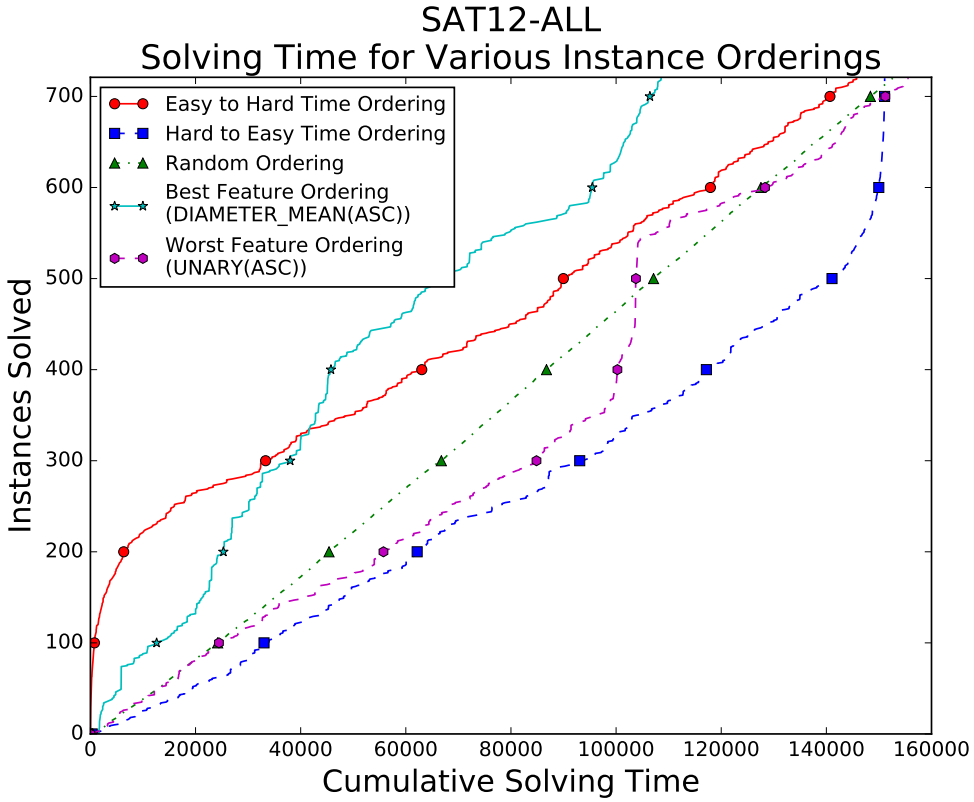


Figure 5.5: SAT12-All: Cumulative runtime graph for best and worst feature orderings with baselines (Avg. over 100 executions).

Figure 5.5 shows the average cumulative solving time for the instance feature orderings, with the best and worst performance as well as the baselines *Random* ordering, *Easy-to-Hard* ordering and *Hard-to-Easy* ordering. Even the worst feature ordering was only 1.9% slower than *Random* while the best feature ordering was 28.9% faster than *Random*. This suggest that we should prefer that instances arrive in almost any feature-

based ordering as the majority of orderings provide an advantage over *Random* ordering and the few orderings that perform worse incur only a relatively small performance penalty. In other words, ordering instances randomly seems to be the worst thing one can do: it is preferable to receive instances in an order that recognises instance size or structure.

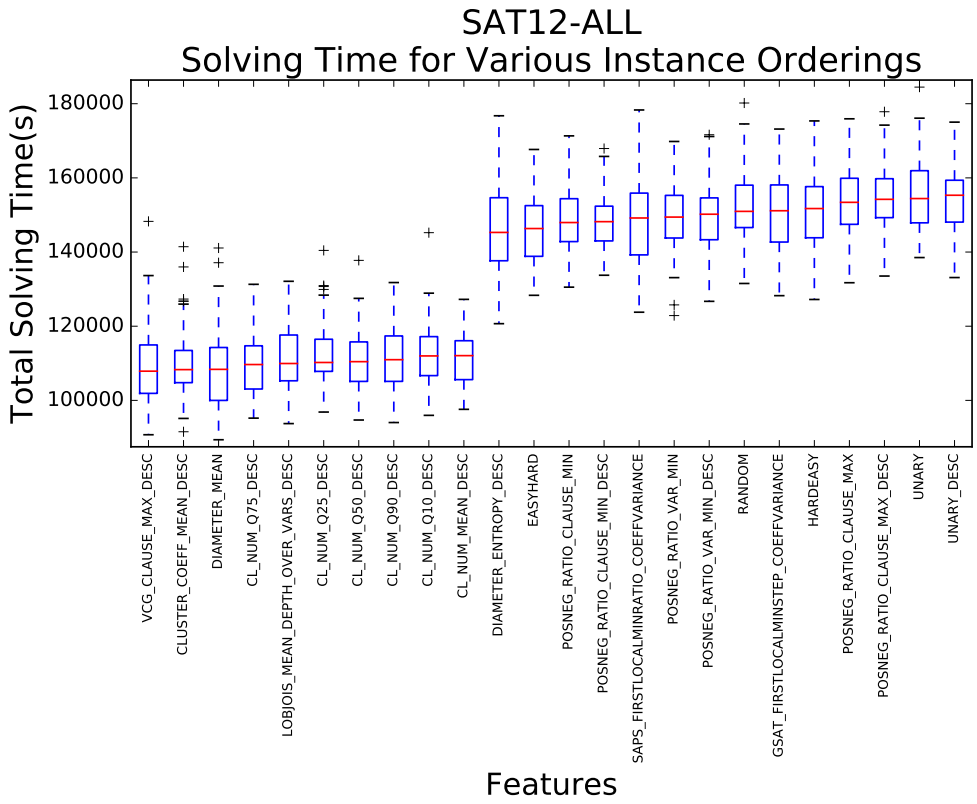


Figure 5.6: SAT12-All: Box-plots of the ten best, ten worst and three baselines (Avg. over 100 executions).

Figure 5.6 shows box plots for the performance of the best ten orderings, the worst ten orderings, and three baselines (*Random*, *Easy-to-Hard* and *Hard-to-Easy*). Features that are appended with `DESC` are sorted in descending order. It is immediately obvious that there is a large gap between the best performing orderings and the worst. This is to be expected given the low p-values seen when statistically comparing these distributions against *Random*. What is somewhat surprising is that the disparity between the performance of the baselines and the worst performing feature orderings is quite small. While *Easy-to-Hard* outperforms *Random*, and *Random* outperforms *Hard-to-Easy* – which agrees with what was shown in previous experiments – all of the baselines are scarcely better than even the worst-performing ordering based on features.

Looking more closely at the features themselves we can see that statistics about the number of learned clauses (sorted descending) dominate the top ten. These clause

learning features are based on a two second run of Zchaff_rand. `DIAMETER_mean` is the mean diameter of the variable graph. `VCG_CLAUSE_max` is a Variable-Clause Graph feature for the maximum clause node degree. `cluster_coeff_mean` is the mean clustering coefficient of the Clause Graph. `lobjois_mean_depth_over_vars` is a DPLL Probing feature which gives an estimate of the search tree size. This suggests that focusing on the constrainedness of instances, and specifically considering more tightly constraint instances first, is a good strategy. Instances of this kind provide greater opportunity to learn, since the relative strength of different solver configurations will be more clearly discernible. If one focuses on instances that are easy for all configurations, then there is little to distinguish them.

Manual inspection of the orderings produced by sorting on these features shows that instances from similar domains tend to cluster together (not shown). This is akin to what we saw in the *Lexicographic* ordering in our previous experiments. However, feature based ordering is more powerful than *Lexicographic* ordering in that it does not rely on instances being within the same folder to group similar instances e.g. crypto instances from the SAT 2007 competition will still appear near crypto instances from the SAT 2009 competition despite not being in the same directory.

We see in Figure 5.6 that the sorting order (ascending or descending) is important for the top performing features. For example `DIAMETER_mean` sorted ascending is the best performing feature taking on average 109k seconds, however `DIAMETER_mean` sorted descending is ranked 145th (132k seconds). It should be noted that both are still statistically better than all baselines. The opposite seems to be true of the poorly performing features whose runtime is close to that of the *Random* ordering. Both ascending and descending orderings of `UNARY`, `POSNEG_RATIO_CLAUSE_MAX` and `POSNEG_RATIO_VAR_MIN` are in the worst ten performing.

Turning our attention to the PROTEUS-2014 dataset again, we find that a large percentage of feature orderings outperform the *Random* ordering. Of the potential 396 orderings (198 features sorted both ascending and descending) 376 have a lower average total solving time. After analysing these results using Student's t-test we find that 310 feature orderings give a statistically significant improvement over the *Random* ordering while only four where statistically worse (`direct_cluster-coeff-mean_DESC`, `support_cluster-coeff-mean_DESC`, `csp_dyn_log_stdev_weight`, `directorder_VCG-VAR-max`). The ordering with the fastest total solving time was 32.1% faster than *Random* ordering while the worst was 6.2% slower. When comparing against *Easy-to-Hard*, we find that 74 feature orderings perform statistically better while 221 are statistically worse.

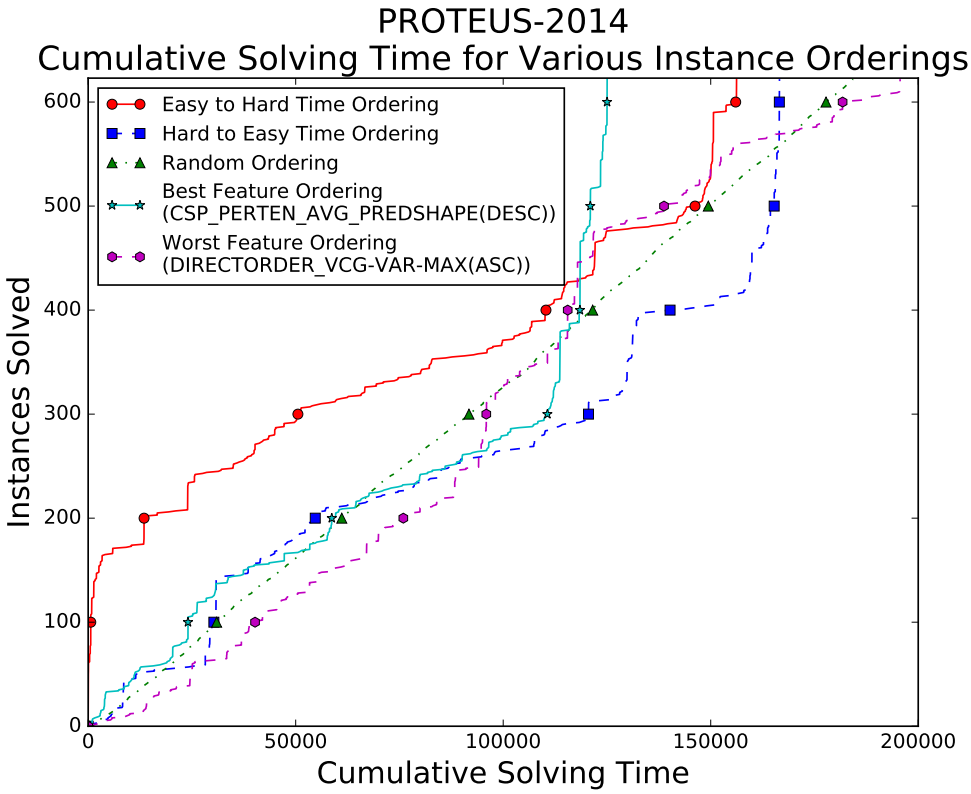


Figure 5.7: PROTEUS-2014: Cumulative runtime graph for best and worst feature orderings with baselines (Avg. over 100 executions).

Figure 5.7 shows the average cumulative solving time for the best and worst feature ordering in addition to the three baselines. The progression for best ordering is interesting in that it follows the gradient of the *Random* ordering for the first 300 instances before solving the remaining 323 instances extremely quickly. This somewhat contradicts the idea that solving the easiest instances first is always desirable. However, it is important to remember that these are static experiments with a fixed pool of solvers. In the dynamic case it may be desirable to solve the easiest instances first and by doing so learn more configurations quickly.

The box-plots in Figure 5.8 show that unlike SAT12-All the ten worst orderings do perform worse than the baselines though not too much worse than the *Random* ordering. The features for the PROTEUS dataset are interesting in that they contain a mixture of CSP and SAT features. The SAT features are replicated for three different encodings used when encoding the CSP instances to SAT (`DIRECT`, `DIRECTORDER`, and `SUPPORT`). These encodings are prepended to the feature names to indicate the encoding used, while CSP features have `CSP` prepended to them. Both the ten best and ten worst performing orderings contain at least one of each type of feature. It is also interesting to note that the PROTEUS and SAT datasets do not agree on the best

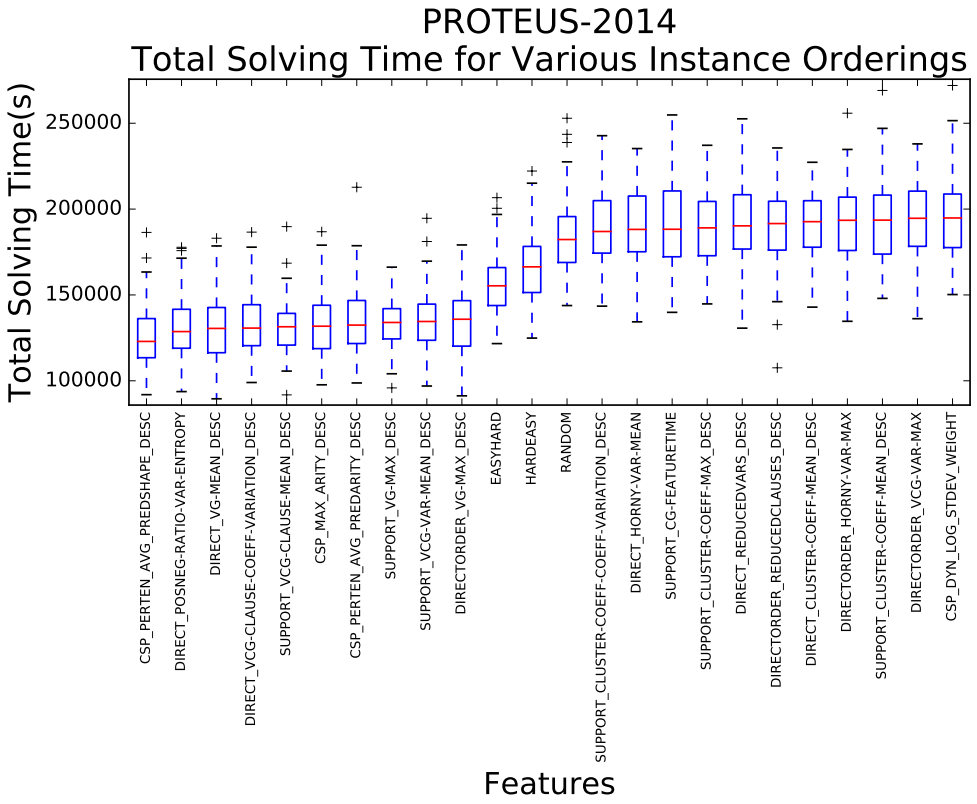


Figure 5.8: PROTEUS-2014: Box-plots of the ten best, ten worst and three baselines (Avg. over 100 executions).

features to order by, in fact CLUSTER_COEFF_MEAN_DESC is the second best feature ordering in SAT whereas it is the third, fourth and sixteenth worst in PROTEUS for SUPPORT, DIRECT and DIRECTORDER encodings respectively.

Variable-graph node degree statistics seem to be important for ordering the PROTEUS dataset. The maximum node degree occurs twice in the ten best orderings (DIRECTORDER_VG-MAX_DESC, SUPPORT_VG-MAX_DESC) while the mean node degree also appears in the top ten (DIRECT_VG-MEAN_DESC). Statistics relating to the Variable-Clause graph occur three times in the top ten (SUPPORT_VCG-CLAUSE-MEAN_DESC, DIRECT_VCG-CLAUSE-COEFF-VARIATION_DESC, SUPPORT_VCG-VAR-MEAN_DESC). These are both interesting since the variable graph and the variable-clause both relate to the constrainedness of the instances, mirroring some of the intuition that lies behind successful variable ordering heuristics for search which prefer more constrained instances, and prefer higher degree variables.

For the CSP features the average predicate shape and arity are important as is the maximum arity (CSP_PERTEN_AVG_PREDSHAPE_DESC, CSP_MAX_ARITY, CSP_PERTEN_AVG_PREDARITY). As with the SAT instances visual inspection of the

orderings produced by these features shows that they tend to group instances from similar domains together (not shown). This, of course, might simply be a consequence of how these problems are modelled.

Looking at the worst ten features to to use for ordering we see that the clause graph features relating to the clustering coefficient perform poorly (SUPPORT_CLUSTER-COEFF-MAX_DESC, SUPPORT_CLUSTER-COEFF-COEFF-VARIATION_DESC, DIRECT_CLUSTER-COEFF-MEAN_DESC, SUPPORT_CLUSTER-COEFF-MEAN_DESC). Two features relating to proximity to Horn formula also appear; the mean and max number of occurrences in a Horn clause for each variable (DIRECT_HORNY-VAR-MEAN, DIRECTORDER_HORNY-VAR-MAX). Similar to the point made previously, the poor performance from using these features reflects the poor performance associated with variable ordering anti-heuristics that prefer less constrained instances.

5.4 Experiments on Non-fixed Solver Configurations

The experiments upto this point in the chapter have used static datasets and ignored the interplay between ordering and the pool maintenance methods, namely configuration removal and generation. While enforcing these limitations was necessary to make the experiments computationally feasible, it does leave some unanswered questions around the impact of ordering on full ReACTR runs. The experiments in this section involve running ReACTR in full which constantly adds and removes configurations to the configuration pool. This differs substantially from the static case we studied above, where a fixed set of solvers were constantly being selected from. Due to the lengthy solving time incurred by full ReACTR runs, these experiments are limited to using a single selection policy. We use the candidate selection policy used in the feature ordering experiments (2TS 2LW 2RAND) for the same reasons outlined previously. The overall objective of these runs is to reduce the mean solving time.

Initially the effect of grouping on solving time was examined. The combinatorial auctions benchmark is an amalgamation of four different types of combinatorial auction instances. By "grouped" we mean that instances are organised by class, and all instances are kept together though within the group they may be ordered differently (*Shuffled*, *Easy-to-Hard*, *Hard-to-Easy*).

Figure 5.9 shows a scatter plot of grouped vs. ungrouped instance solving times for various orderings. All points occur above the identity line which means that ungrouped performance improves upon that of grouped in every case. This result is somewhat unexpected especially considering that the lexicographically-ordered instances (a type

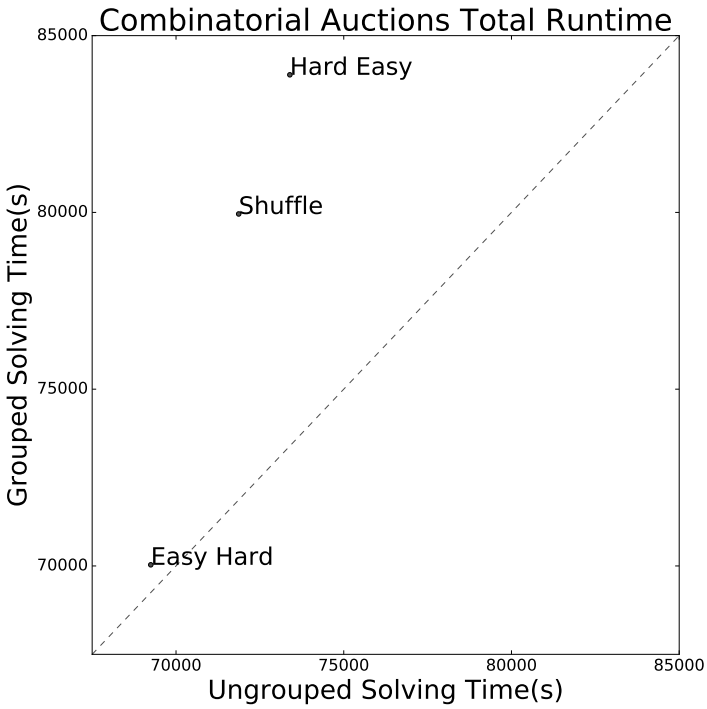


Figure 5.9: A scatter plot showing the total solving time for grouped vs. ungrouped instances on the Combinatorial Auctions benchmark (Avg. over 10 ReACTR executions).

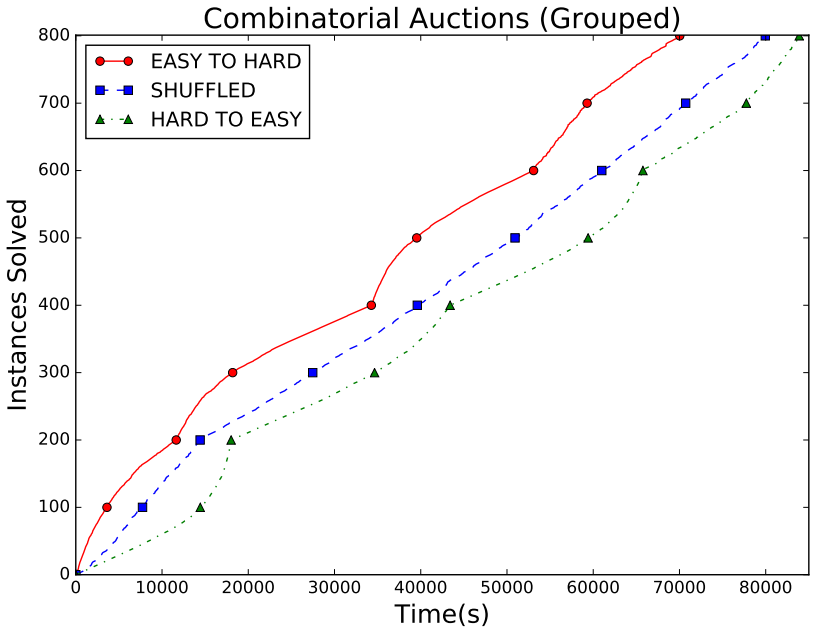


Figure 5.10: Instances Solved vs. Solving Time for grouped instances of the Combinatorial Auctions benchmark (Avg. over 10 ReACTR executions).

of grouping) had the lowest solving time in the static experiments. One possible explanation for this is that a type of over fitting occurs when the configurator only encounters instances of a single type. Due to the fact that all instances are of a single type

initially, specialised configurations may beat more generally applicable configurations and cause them to be removed. This hypothesis is supported by the trajectory of the grouped and shuffled instances in Figure 5.10. Here, the solving time for the initial group looks promising with a steep incline in the plot but after changing groups at 200 instances the slope becomes flatter, denoting slower solving time. Student's T-Test shows that both the Ungrouped Hard-to-Easy and Shuffled orderings outperform their Grouped counterparts (P=0.006 and P=0.054 respectively). The Easy-to-Hard results were not found to be statistically better (though this could be due to the relatively small number of runs).

The box plots in Figure 5.11 provide further evidence that grouping is not beneficial during dynamic ReACTR runs. Interestingly, not only do all grouped runs perform worse than their ungrouped counterparts but they also exhibit a much larger spread in terms of solving time.

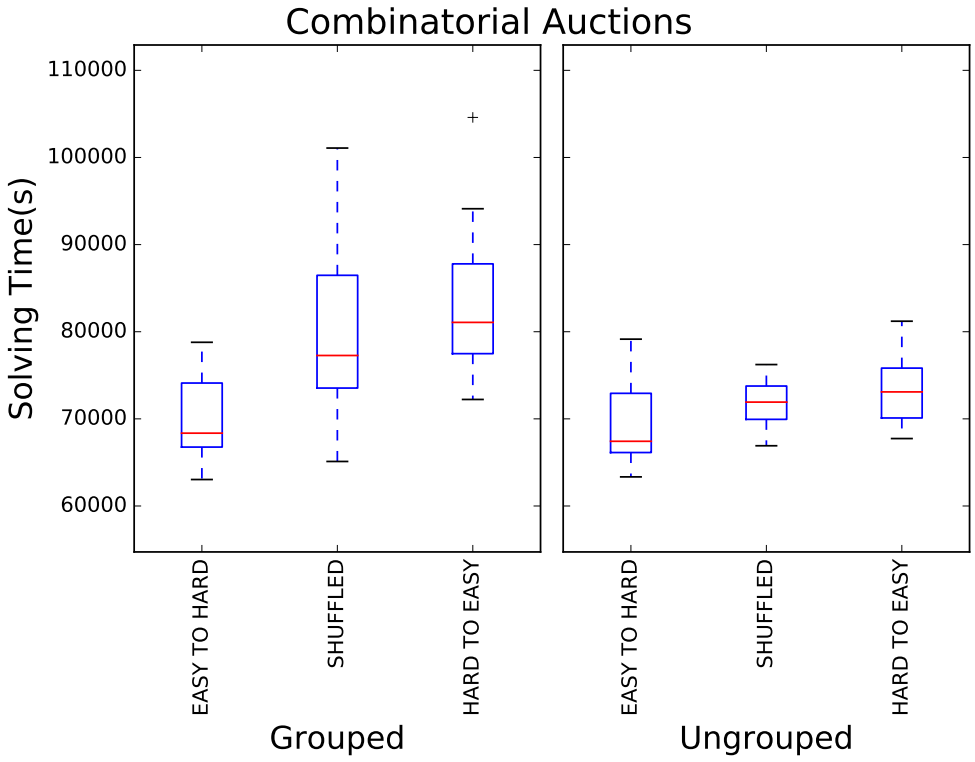


Figure 5.11: The total solving time for various orderings of the Combinatorial Auctions benchmark both grouped and ungrouped (Avg. over 10 ReACTR executions).

Figure 5.11 also shows that ordering instances by Easy-to-Hard results in the fastest solving time regardless of grouping or not; Shuffled instances are in the middle in terms of solving time, and Hard-to-Easy instances take the longest to solve. This agrees with what would be expected intuitively. At the start the configurator has not had a chance to find any improvement and so solving hard problems is detrimental to the overall solving

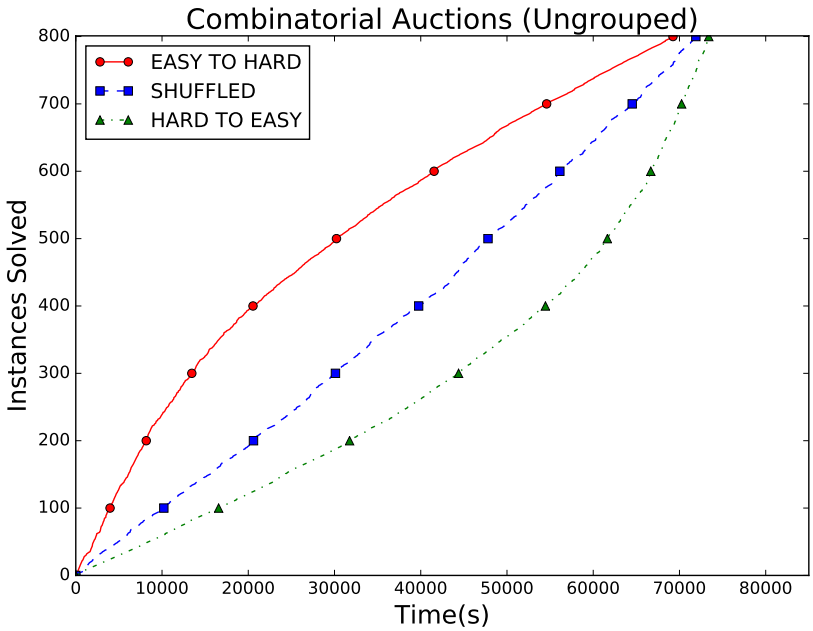


Figure 5.12: Instances Solved vs. Solving Time for ungrouped instances of the Combinatorial Auctions benchmark (Avg. over 10 ReACTR executions).

time. Figures 5.10 and 5.12 show what is happening more clearly: Hard-to-Easy solves fewer instances early on and, as such, the configurator learns less while also solving the instances more slowly because the configurations have not improved yet. Easy-to-Hard is able to solve the easy instances first which even the default configuration, which is used due to warm starting, should solve relatively quickly. Because the instances at the beginning of the Easy-to-Hard ordering are less challenging, ReACTR is also able to solve more in a short amount of time. This allows the configurator to learn better configurations much faster than if it were solving hard instances. By the time the solver has reached the more difficult instances at the end of the Easy-to-Hard order of instances it has learned multiple good configurations to make solving difficult instances much quicker.

5.5 Chapter Summary

In this chapter we investigated the effect instance ordering and candidate selection have on the real-time algorithm configurator ReACTR. We demonstrated that both the selection procedure used to select configurations and the order which instances arrive in have a significant impact on ReACTR's performance.

Furthermore we showed that instance ordering and the selection procedure used are linked. Some candidate selection procedures are better suited to certain instance

orderings than others. Choosing the correct selection procedure can lead to marked improvements in configurator performance.

The efficiency of ReACTR when configuring over streams various instance orderings was also examined. Using two static datasets (where no configuration replacement occurs) in different domains we showed that ordering based on nearly any feature value was better than doing so at random. We also evaluated difficulty and group-based orderings using full ReACTR configuration runs. Somewhat surprisingly in this case grouping appeared to hamper the performance of the configurator. We believe this is due to overfitting that occurs when the configurator encounters only one type of instance for a long time and discards more generally applicable configurations from its pool.

Chapter 6

Configuration Pool Maintenance

Summary. *This chapter considers how the internal pool of configurations in ReACTR is maintained. In order for the real-time algorithm configuration system to function effectively it must evaluate configurations and remove those that are subpar as quickly as possible. However, there is a tradeoff to be made; quickly removing configurations comes with a greater risk of inadvertently removing strong configurations. In this chapter we discuss various removal strategies and metrics that aim to maximise efficiency while minimising the risk of unwittingly removing good configurations. Every configuration that is removed must be replaced by another. The quality of the replacement configurations generated dictates the level of improvement that the configurator will be able to achieve. For this reason the procedures used to generate new configurations are of the utmost importance. We show that a simple approach based on genetic algorithms can provide good quality configurations while maintaining a small enough computational overhead to be practically applicable to the real-time system. Additionally, we demonstrate how model-based techniques can be leveraged to reduce the configuration space for more efficient search.*

6.1 Removal Strategies

The rate an online configuration system is able to evaluate new configurations is predominantly determined by the removal strategy it adopts. As the configuration pool is fixed in size, weak configurations must be removed in order to make room for new candidate configurations. Increasing the number of candidates evaluated makes it more

likely for the system to encounter superior configurations. For this reason the rate at which subpar configurations are removed is directly linked to the configuration search speed and the success of the real-time configuration system.

While rapidly removing inferior configurations allows the configuration search to progress quickly, it is important not to remove viable candidates or strong incumbents in this haste. In an ideal situation we would be able to evaluate each configuration only once and decide definitively whether it will solve future instances quickly or not. This is the case when optimising parameters for deterministic functions such as artificial optimision test functions (Ackley, Branin etc.). However, in a general algorithm configuration scenario this does not hold true; various instance encodings and stochastic solver decisions produce a distribution of different runtimes for each instance. This is further compounded when looking at multiple instances that can have different properties and levels of difficulty. The number of quality configurations in a configuration space is typically eclipsed by quantity of poor configurations. Therefore it is of the utmost importance that when good configurations are discovered that they are not removed prematurely. Any removal strategy adopted for a real-time configuration system must be robust enough to evaluate configurations quickly with a high degree of accuracy. Therefore, there is a careful balance to be struck between quickly removing underperforming configurations without endangering those that contribute to the success of the configurator.

In the rest of this section we look at a number of potential removal procedures from simple to complex. Section 6.1.1 looks at simple numerical approaches while in section 6.1.2 we demonstrate how the ranking systems, used previously for candidate selection, can be applied to rank and remove configurations from the candidate pool.

6.1.1 Simple Numerical Methods

6.1.1.1 Individual Win/Loss Ratio

One simple removal strategy is to look at the ratio of wins and losses. This can be done at the individual configuration level or using a pairwise comparison against another configuration. Considering the individual case, the ratio is calculated as the number of configuration races that that configuration has finished fastest (assuming a runtime minimisation objective, as with all our experiments) divided by the number of runs in which that configuration has participated. This will give a number between 0 and 1 where 1 is a perfect win rate. This approach has the advantage that it is extremely fast to compute and easy to interpret.

There are, however, also a number of disadvantages to this method of removal. All instances (races) are considered equal with no advantage given to more recent wins. This may not necessarily be a bad thing: if all instances are known to be homogeneous then recent wins are no more important than older wins. However, if the instances drift over time, as is common in online stream processing scenarios, then weighting more recent instances more heavily can be an advantage.

Setting this aside, the simple win ratio approach to removal also has another challenge in that it must be manually tuned. Deciding at what cut-off point to remove a configuration is not as simple as choosing the ratio because that ratio does not tell the whole story of how much an instance has been tested. For example, a configuration that has won one of the 2 races it has competed in and another configuration which has won 400 of the 800 it has competed in both have win ratio of 0.5 however we can be much more confident in our estimation of the latter's ability because there is a much larger sample of evidence. For this reason the win ratio removal strategy should be used with an additional criterion that specifies the minimum number of runs used before removal. Requiring a fixed minimum number of evaluations contradicts the earlier stated goal of removing under-performing configurations as quickly as possible.

6.1.1.2 Pairwise Win/Loss Ratio

Another simple ratio based removal strategy is to compare all pairs of candidates in the configuration pool that have competed against one another a sufficient number of times. The inter-configuration win-loss ratio is then used to determine if one candidate is dominating another. Adopting this pairwise approach rather than the individual approach helps remove configurations that are superseded by others rather than relying on the individual ranking.

The procedure for determining a *weak* paramaterisation p to be removed is as follows:

1. been beaten by another parameterisation, p', at least m times and
2. the ratio of the number of times p' has beaten p to the number of times p has beaten p' is greater or equal to r.

The former criterion ensures that parameterisations are not removed without being given ample opportunity to succeed. The latter criteria ensures that the domination of one parameterisation over another is not just due to random chance. This is the removal strategy that is used in the initial version of our real-time algorithm configuration system, ReACT.

6.1.2 Ranking Systems

While conceptually simple, numerical methods provide a low overhead method for identifying weak configurations. There is much more room for improvement by using more involved methods. In Chapter 5 we saw that ranking methods originally designed for ranking players in games can provide accurate estimations of a configuration's merit. Previously we used these techniques as part of the selection procedure in order to identify promising candidates to compete. Conversely, such methods can also be used to both identify subpar configurations and gauge the level of confidence our system has in the rank assigned. By using this information as part of the removal mechanism it is possible to more accurately identify which configurations can safely be removed. As an added bonus, the system already uses these methods to compute the ranking for selection purposes so there is no increase in computational resources required. Here we focus on the two most promising ranking techniques: Glicko and TrueSkill.

6.1.2.1 Glicko

The qualities which Glicko possesses that make it suitable for selecting configurations are also very useful when deciding which configurations to purge from the pool; namely its quality ranking and confidence estimate (ratings deviation). However, it also faces the same drawbacks: it was designed as an improvement to the Elo rating system and as such only handles two player games. Pairwise comparisons allow the ranking of multiple configurations but this is a workaround rather than using the system as designed. Using this workaround also greatly increases the amount of computation required as all pairwise combinations of configurations must be evaluated. In the interest of brevity we will not repeat the details of the Glicko system, instead we refer the reader to section 5.1.3 for a full description of the Glicko and Glicko 2 ratings systems.

6.1.2.2 TrueSkill

Similar to Glicko, TrueSkills ranking and confidence intervals provide a powerful mechanism for removing weak configurations quickly and safely (high confidence in low performance). TrueSkill has a number of additional benefits over Glicko as well. It is designed with games featuring 2+ players in mind from the outset. Also, as it was originally devised by Microsoft for ranking players in large multi-player online video games the calculations are fast and able to scale massively.

Finally, and most importantly, TrueSkill outperforms Glicko. In Chapter 5 we outline the results of an experiment where we implemented the ReACT framework using both ranking systems and evaluated them against one another. This comparison showed us

that TrueSkill's system, which allows it to compare multiple configurations simultaneously, is superior for our purposes. This allows us to have confidence in selecting TrueSkill as the ranking algorithm to select weak configurations for removal.

6.1.2.3 TrueSkill Thresholds

With a powerful ranking method selected to identify configurations for removal, it is important to correctly configure its thresholds in order to achieve the best performance. At this juncture it is again important to reiterate the trade-off that must be made between fast removal and confidence. We can quickly purge many configurations, however, increasing the risk of inadvertently removing a good configuration. In order to avoid this we examined three different methods of identifying the cut-off point at which to remove configurations: fixed score removal, a quantile-based approach, and using TrueSkill's built in ability to calculate win probabilities.

Fixed Score Removal The most straightforward method of defining removal thresholds is to specify fixed TrueSkill skill and confidence values. When the system is confident that a configuration's skill has dropped below this cut-off the configuration is removed. The issue with this is that it requires the user to have some intuition about appropriate values or conduct trials to identify appropriate values. Another challenge with this approach is that the system does not adapt to changes in the score distribution for example score inflation due to better configurations being added.

Example 6.1.1. For example assume the configuration pool consisted of the following four configurations and their associated TrueSkill scores:

- *A* - $\mu = 24.3$, $\sigma = 6.9$
- *B* - $\mu = 27.8$, $\sigma = 3.2$
- *C* - $\mu = 29.5$, $\sigma = 1.8$
- *D* - $\mu = 22.7$, $\sigma = 2.5$

Given a fixed score removal threshold of $\mu = 25.0$, $\sigma = 5.0$ we would select configuration *D* for removal. Notice that although configuration *A*'s score is below the threshold, we are not confident enough in this rating to remove it yet.

Quantile-Based Removal Alternatively we can specify the quantile cut-off and remove configurations that fall below this. This method dynamically adjusts the threshold to account for changes in the scoreboard skill distribution. However, this method also requires a confidence level to be specified which may be difficult to infer in advance.

Example 6.1.2. This time we examine a quantile-based removal scheme which remove configurations with a threshold of the middle quantile (median) and with confidence $\sigma = 7.0$. We use the same set of configurations and scores as the previous example. In this case the removal threshold is $mu = 26.05$ so we remove configurations *A* and *D*.

Win-Probability Removal While the empirically derived skill and confidence values used for candidate removal in ReACTR work well, TrueSkill offers us a more widely applicable approach requiring little configuration. Using the μ and σ values provided by TrueSkill it is possible to calculate the win probability between two configurations. As TrueSkill skill is modelled as a normal distribution it is possible to calculate the overlap between the distributions. Using the cumulative distribution function of this overlap we are able to calculate the expected win probability of each configuration.

Example 6.1.3. Our final removal metric example looks at a win-probability removal scenario. Again we use the same set of instances and scores as the previous examples and fix our win-probability threshold at 0.2. We use TrueSkill's built-in method to calculate the win probability of the other configurations against the incumbent (*C*) with the following results:

- *A* - 0.29
- *B* - 0.4
- *D* - 0.15

Configuration *D* has only a 15% of winning against configuration *C* and so it will be removed.

Evaluation To evaluate each of these methods we opt to use data from six common datasets covering a range of solvers and optimisation domains. This data comes from algorithm configuration runs of ACLib used to train surrogate models [HLF+14, ELH+18]. These are:

- lingeling_circuitfuzz
- clasp_queens
- cplex_rcw
- minisat_randomk3
- probsat_7sat90
- cplex_regions200

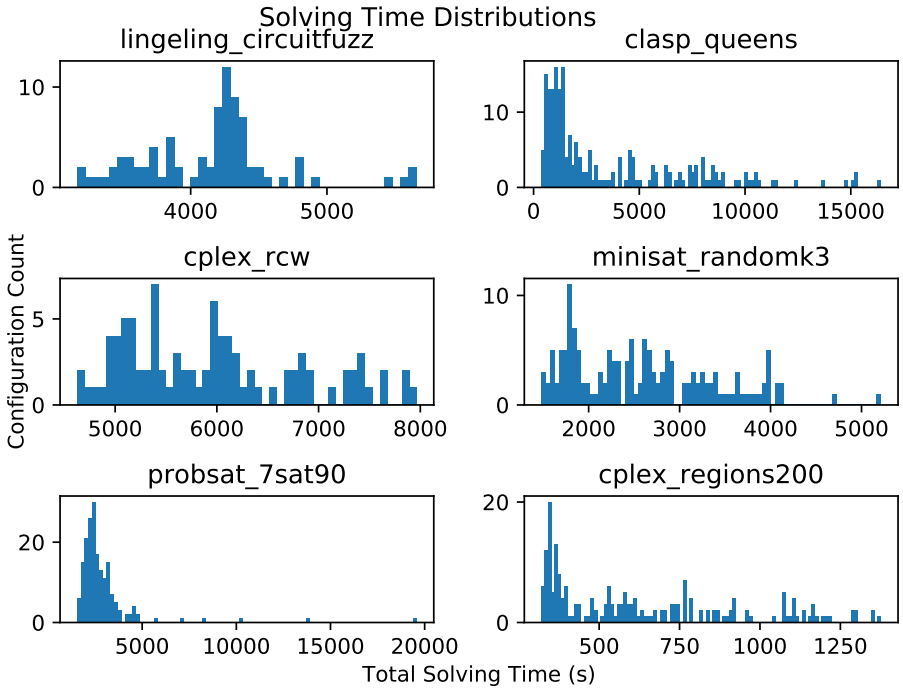


Figure 6.1: Runtime distribution of instances used.

As this data is collected from offline configuration runs many instances are only solved by a small number of configurations. Online configuration runs may introduce a configuration at any point while processing stream, as such we select a subset of 100 instances from each dataset which have been solved by a large number of configurations to ensure a ready supply. Where there are still insufficient configurations for the run, for example when using very aggressive removal thresholds, we supplement this supply with dummy configurations which are recorded as timing out in every race in which they participate. To avoid a glut of real configurations early on followed by many dummy configurations we introduce real configurations at a rate proportional to their availability and the number of instances left to be solved (rate = remaining real configs÷remaining instances). Figure 6.1 shows the distribution of total solving times for each configuration (excluding dummy configurations).

Computing the outcome of these races inexpensively by processing existing run data allows us to perform 100 runs of each threshold instantiation. For each run the instance ordering remains identical but we alter the configuration starting pool selection, and the order in which new configurations are supplied leading to completely distinct races. We use similar ReACTR parameters to our other experiments, namely pool size 30, race size 6 and epsilon-greedy selection with a single good configuration and 5 random; note that this is slightly different from the 2 good and 4 random used in Chapter 3.

We test multiple settings for each threshold method. In the case of *fixed score* we evaluate μ in the integer range 24 to 27 (inclusive) as well as high and low confidence level for σ, 3 and 6 respectively; the default TrueSkill starting value in ReACT is $\mu = 25$ and $\sigma = 8.3\dot{3}$. For the *quantile* experiments we use the three quartiles (0.25, 0.5, 0.75), and the same high and low confidence values as with *fixed scores*. Finally, we assess values in the range 0.1 to 0.4 in increments of 0.1 for our *win-probability* thresholds.

Figure 6.2 shows the results of these experiments. The columns in Figure 6.2 show different threshold methods: *Fixed Score*, *Quantile*, and *Win Probability*, respectively. Each row displays a different dataset. The X-axis shows the number of configurations evaluated while the Y-axis represents the total solving time in seconds. For clarity of comparison the scale of the axes is fixed across all subfigures. Table 6.1 presents summary statistics for total solving time in seconds (*Time*) and number of configurations processed (*Configs*) for these experiments.

Quantile-based removal methods appear to perform best overall in terms of solving time. Settings that favour more aggressive removal, i.e. Q_2 and Q_3, are most effective. It is not as easy to select which confidence level to apply as this seems largely dependent on the dataset. Looking at the configuration processing speed it is clear that quantile-based methods tend to process fewer configurations than other removal methods.

Fixed score method's solution times are competitive with quantile-based methods when the correct parameters are used. Fixing $\mu = 26$ and $\sigma = 3$ provides the best overall solving performance across datasets. It is important to note here that these experiments are run on a relatively small stream of 100 instances, on larger streams it is likely that the score distribution will drift and cause fixed values to become suboptimal.

Finally, and somewhat surprisingly, removal using the calculated win-probability performs considerably worst across all datasets. Removal methods based on the win probability exhibit the two extremes in terms of processing speed also. A win probability of 0.4 leads to an excess of removals. Given that 6 configurations are evaluated in each race (with the incumbent being one of those) and there are 100 races, we would expect an absolute maximum of 500 configuration evaluations which this exceeds. This implies that even configuration that won a race were removed. On the other end of the spectrum a win probability of 0.1 is too cautious and fails to evaluate any configurations beyond the starting pool.

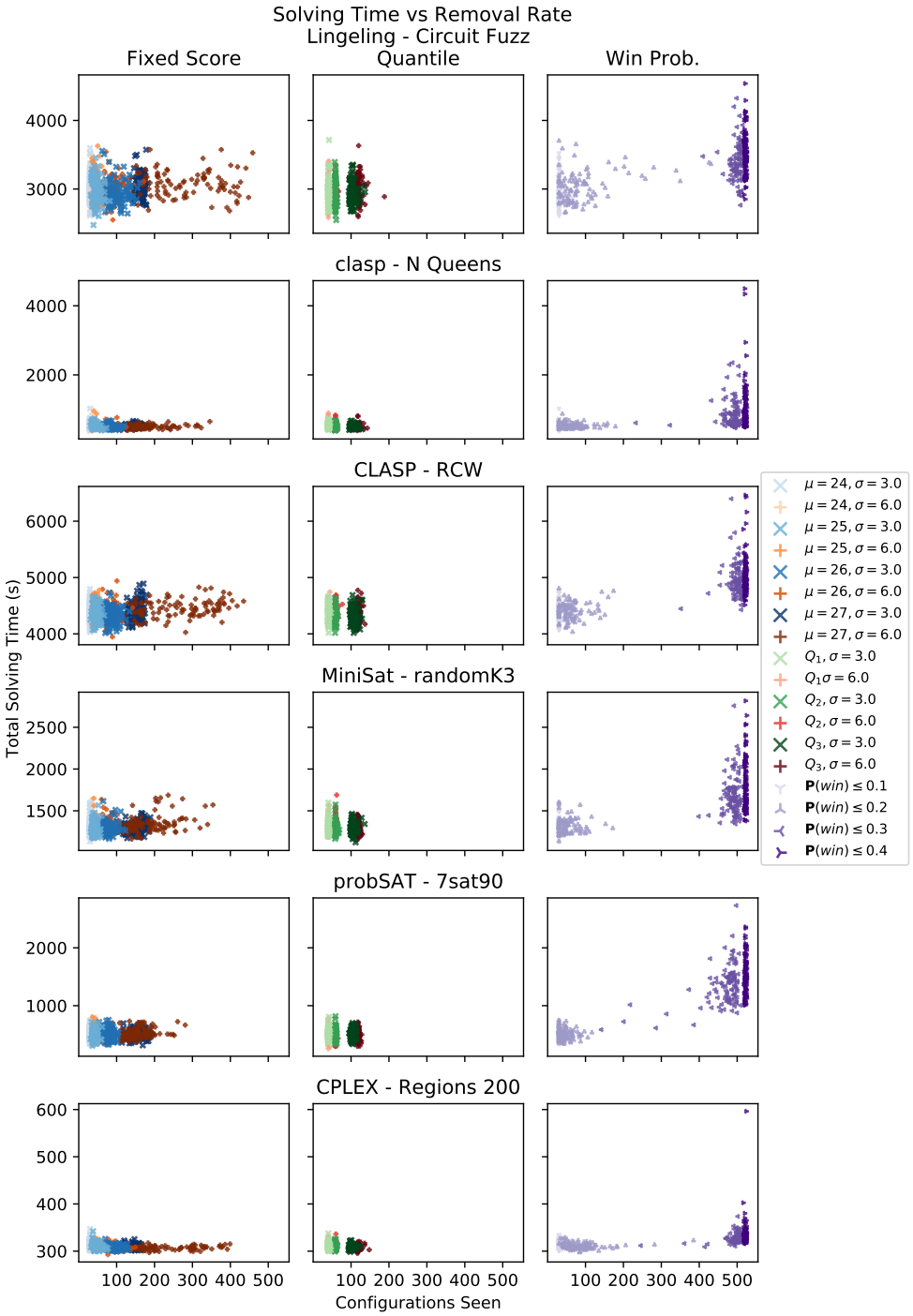


Figure 6.2: Solving time vs. configurations processed using various TrueSkill thresholds.

Table 6.1: Summary statistics for total solving time(s) and configurations processed using different thresholding methods and settings.

			Lingeling Circuit Fuzz		clasp N-Queens		CLASP RCW		MiniSat randomK3		probSAT 7sat90		CPLEX Regions 200	
			Time	Configs	Time	Configs	Time	Configs	Time	Configs	Time	Configs	Time	Configs
Method	Threshold	Dist.												
Fixed Score	$\mu = 24, \sigma = 3.0$	Avg.	3011	30	553	30	4395	30	1314	30	520	30	315	30
		Std.	185	1	93	1	159	1	92	0	90	1	8	1
	$\mu = 24, \sigma = 6.0$	Avg.	2997	35	548	35	4366	35	1308	35	507	35	313	35
		Std.	179	1	81	0	146	1	85	1	89	1	7	1
	$\mu = 25, \sigma = 3.0$	Avg.	2962	48	522	45	4355	46	1295	43	492	43	312	48
		Std.	178	10	68	7	150	7	74	6	89	5	7	9
	$\mu = 25, \sigma = 6.0$	Avg.	3012	46	535	45	4399	45	1298	43	518	42	311	48
		Std.	192	10	81	7	152	7	74	5	94	5	5	9
	$\mu = 26, \sigma = 3.0$	Avg.	2947	101	501	85	4324	88	1293	80	496	77	309	92
		Std.	169	24	54	14	140	15	70	14	91	12	5	18
	$\mu = 26, \sigma = 6.0$	Avg.	2990	89	510	80	4377	85	1304	77	506	75	309	85
		Std.	159	19	61	15	160	13	80	11	79	9	5	16
	$\mu = 27, \sigma = 3.0$	Avg.	3032	164	519	146	4437	151	1296	149	509	155	310	146
		Std.	159	11	45	14	157	14	66	22	74	21	4	13
	$\mu = 27, \sigma = 6.0$	Avg.	3080	275	501	181	4454	268	1349	189	533	159	307	220
		Std.	180	94	54	51	150	83	107	56	93	34	4	81
Quantile	$Q_1, \sigma = 3.0$	Avg.	2954	41	531	40	4380	40	1304	40	519	40	313	41
		Std.	164	1	72	1	139	1	77	1	80	1	6	1
	$Q_1, \sigma = 6.0$	Avg.	2984	40	546	40	4413	40	1312	40	512	40	313	40
		Std.	169	1	90	1	126	1	79	1	103	1	7	1
	$Q_2, \sigma = 3.0$	Avg.	2931	58	520	58	4309	58	1295	59	510	59	310	58
		Std.	162	1	61	2	119	2	66	2	88	1	5	1
	$Q_2, \sigma = 6.0$	Avg.	2961	60	517	59	4390	60	1310	60	493	59	312	60
		Std.	133	1	80	1	132	2	82	1	83	1	6	1
	$Q_3, \sigma = 3.0$	Avg.	2940	104	506	104	4327	106	1291	106	503	104	308	102
		Std.	153	8	50	6	137	7	62	6	83	4	4	6
	$Q_3, \sigma = 6.0$	Avg.	2989	121	501	119	4365	119	1299	119	492	119	308	119
		Std.	178	9	63	4	135	4	69	3	86	2	5	5
Win Prob	$\mathbf{P}(win) \leq 0.1$	Avg.	2987	31	555	31	4412	30	1316	30	518	30	315	32
		Std.	198	4	91	5	144	2	93	2	88	0	7	5
	$\mathbf{P}(win) \leq 0.2$	Avg.	3034	93	527	65	4401	63	1317	53	509	46	311	88
		Std.	219	79	76	32	148	32	83	26	92	19	6	51
	$\mathbf{P}(win) \leq 0.3$	Avg.	3398	502	857	484	4948	498	1683	489	1342	463	328	495
		Std.	274	15	396	33	291	21	235	17	342	60	10	32
	$\mathbf{P}(win) \leq 0.4$	Avg.	3524	522	1044	521	5046	522	1818	522	1535	522	337	521
		Std.	309	2	657	2	341	2	304	2	310	2	29	2

6.2 Configuration Generation

While efficient evaluation and removal are extremely important for the success of the ReACTR real-time algorithm configuration system, it is the configurations themselves that are the ultimate goal and dictate how long instances will take to solve. An algorithm configurator is only as good as the configurations it generates. Therefore the higher the quality of the configurations generated, the better the performance of the configurator.

In an ideal configuration scenario every new configuration introduced would outperform the previous incumbent resulting in constant improvement. Unfortunately this is not a possibility without some form of *oracle* that can somehow divine the quality of a configuration. If an *oracle* were available we could instantly jump to the globally best configuration or best configuration per instance. In reality the process of generating new instances is one of trial-and-error. However, this can be conducted in an intelligent manner such that previous knowledge and experience can guide and direct the generation procedure to areas where improvement is more likely to be realised. Another challenge is to decide how much time to spend exploiting this previous knowledge and how much time to spend exploring different, diverse, solutions. This balancing act, though it sounds simple, is actually integral to the success of the generation procedure. Allocating an excess of time to exploring solutions similar to previous good configurations may result in getting trapped in local optima. On the other hand, too much time spent exploring leads to a random sampling of points in the configuration space in the hopes of landing in a promising region. This methodology fails to utilise our existing knowledge in order to systematically and efficiently search in areas likely to result in improvement.

Finally, it is important to note that not all parameters are created equal. In many cases a small handful of correctly set parameters can result in the lion's share of the improvement [HHL13, HHL14, FH16]. By identifying and focusing on this smaller subset of parameters rather than the full spectrum it is possible to dramatically reduce the configuration space. In the same way that adding parameter options to a configuration causes an exponential increase in the size of the configuration space, removing (or not considering) parameters allows for an exponential shrinking of the configuration space resulting in a faster search procedure. This idea is explored in subsection 6.3.

6.2.1 Exploration and Exploitation

Previously, in Chapter 5, we encountered the exploration and exploitation dilemma when deciding which configurations to race. Generating new configurations presents us with a similar challenge: should we choose untested parameter values and combinations,

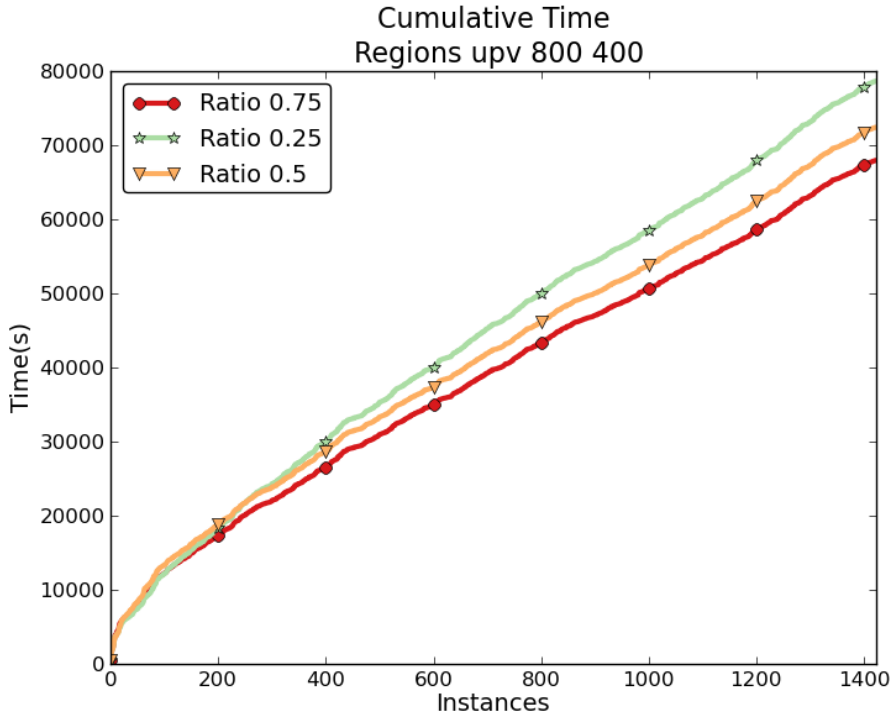


Figure 6.3: Cumulative solving time for ReACTR with different exploitation ratio settings.

or recombine existing promising parameter settings?

To explore this trade-off we allow for a variable to control the balance of exploration to exploitation, or the percentage of configurations generated at random against those engineered to perform well. Figure 6.3 shows the effect of varying the ratio of exploitation on cumulative solving time for ReACTR configuring CPLEX [IBM14] for arbitrary combinatorial auctions instances. This experiment uses the ReACTR default instantiation of randomisation for exploration and crossover as the exploitation mechanism (described in Sections 6.2.2 and 6.2.3 respectively). From this graph, we can see that, at least in the relation of these two methodologies, it is better to actively exploit existing knowledge, generating a greater proportion of configurations using crossover.

Despite this, ReACTR's default setting weighs exploration and exploitation equally in order to avoid stagnation in the configuration pool. It is also worth mentioning that although fixing a single value in advance is conceptually simple, the optimal ratio is almost certainly dynamic given that it relates to the diversity within the pool, the quality of the current configurations, and the progress of the search. This presents an interesting avenue for future research.

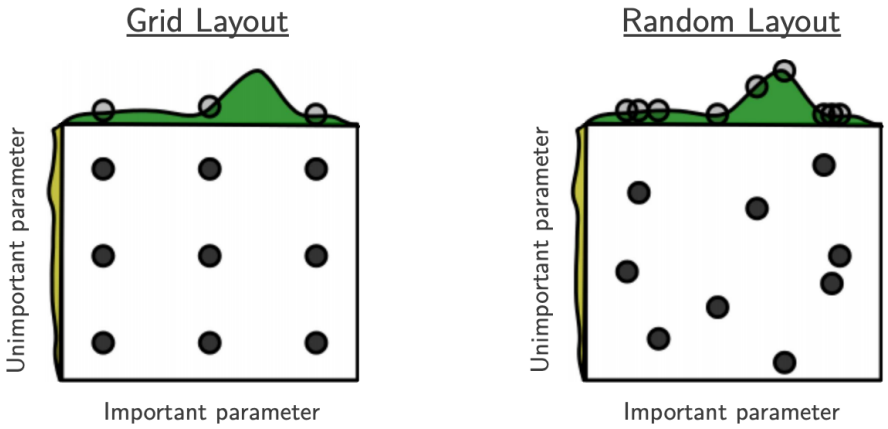


Figure 6.4: Grid vs Random Search. Credit [BB12].

6.2.2 Randomisation

One of the most intuitive and inexpensive methods of generating configurations is to sample uniformly at random from the configuration space. While this approach appears overly simplistic at first, it has been shown to match and outperform traditional non-heuristic based sampling approaches such as grid search [BB12]. Random sampling works particularly well in high dimensional spaces, such as those typically encountered in algorithm configuration, as these often have a low effective dimension [BB12, FH16]. A function, with a large number of parameters, is said to have low effective dimension if it can be approximated by another function with a much smaller number of parameters. Generating configurations in this manner is entirely explorative as the search is not guided in any way.

Bergstra and Bengio [BB12] give the example in Figure 6.4 where some function f with x and y inputs can be approximated by another function g with just x, more formally, $f(x, y) = g(x) + h(y) \approx g(x)$. The figure demonstrates that due to the structure of grid search that the important dimension, $g(x)$ is effectively only sampled 3 times during 9 total evaluations, on the other hand, all evaluations on this space using random sampling provide distinct results for $g(x)$.

Due to its effectiveness and simplicity, this random sampling approach is what was used exclusively to generate configurations in our initial instantiation of the ReACT framework. When generating a new configuration, ReACT sampled a random setting for each parameter from the allowed values while taking care to correctly resolve any conditional parameters. By generating configurations in this way, ReACT is able to explore the large multidimensional configuration spaces quickly. This approach allowed ReACT to achieve strong results, improving over solver defaults and closing the gap

between it and offline configurators. ReACTR also uses this approach to generate configurations however these are interwoven with configurations generated by genetic algorithms in order to balance exploration and exploitation (SMAC adopts a similar strategy for the same reasons [HHL11]).

6.2.3 Genetic Algorithms

Genetic algorithms (GA) are a class of population-based optimisation algorithms which take inspiration from Charles Darwin's theory of natural selection. GAs are a subset of a wider class of Evolutionary Algorithms where potential optimisation problem solutions are modelled as chromosomes. Each chromosome is made up of multiple constituent genes and is scored according to some fitness function. Over a number of generations the population of chromosomes evolves by replacing some or all of the current population with new offspring created by recombining existing chromosomes. Chromosomes with a higher fitness value are more likely to participate in the recombination step and as such their genes are more likely survive, so called "survival of the fittest" in Darwin's work [Dar59].

GAs are flexible in that they can be used to optimise a large variety of black box optimisation problems. This flexibility comes from being agnostic to the shape of the search space unlike some other optimisation methods which rely on gradients. GAs have gained wide acceptance and there is an abundance of literature outlining improvements and real world applications ranging from aerial design to modelling *E. Coli* cultivation [dAP12, RF12]. In fact, as discussed previously, genetic algorithms have even been used very successfully in the area of offline algorithm configuration [AST09b, AMS+15]. Our approach, outlined in the following sections, is tailored to the real-time configuration problem and as such differs from GGA(++) in some important areas.

6.2.3.1 Problem Encoding

In order to make use genetic algorithms to find improving configurations we must first define a problem encoding and fitness function. There are a number of methods for encoding problems to be optimised by means of genetic algorithm [SP94]. Some of the more commonly used methods include binary encoding, tree encoding, and value encoding.

Binary encoding is traditionally the most common method and works well for binary values e.g. include this edge in a graph or not. However, when each value can assume multiple values binary encoding requires that these are converted to binary vectors. In

Table 6.2: An illustration of a drawback in the binary encoding scheme.

Parameter Name	Binary Encoding	Fitness Score	Propagation Probability	Binary Prop. Probability
A	00	4.4	0.169	0.232
B	01	4.8	0.185	0.122
C	10	12.6	0.485	0.422
D	11	4.2	0.162	0.224

many cases this encoding is not a natural representation of the problem and can lead to a number of issues.

Example 6.2.1. Binary encoding may require additional checks to ensure the gene is a valid value. For example, to encode a value with three options, binary encoding would require a 2-bit vector but a 2-bit binary vector can encode upto four values. Therefore it is necessary to apply additional logic in order to forbid this illegal choice.

Example 6.2.2. Another reason to opt against binary encoding is that it adds relationships between options where there are none. In this case that would result in one parameter value being disproportionately reproduced because it shares a single bit with a different strong parameter value. To give a concrete example, consider the four following categorical parameters, A-D, with associated binary encoding and scores:

Assuming parameters are added to the pool using roulette wheel selection we would expect the probability of each parameter value propagating to be that in "Propagation Probability" column of Table 6.2 [SP94]. However, though the binary encoded parameters are added to the pool in the same ratio, the probability of the values propagating differs, column "Binary Prop. Probability" shows this. This occurs because the probability of each bit in the binary encoding propagates independently of one another.

Tree Encoding is typically used in genetic programming [Kum13] in order to evolve programs where a certain structure must be maintained. GGA uses a tree encoding in order to link related conditional parameters to one another [AST09b]. There is no technical reason that this form of encoding could not be implemented in the ReACT framework.

However, in our ReACTR work we instead opt to use a direct value encoding [Kum13]. With this form of encoding each variable is selected uniformly at random from the set of of possible valid values. For categorical values this is as simple as selecting from a list of supplied values while both integer and continuous values are selected uniformly at random from within a range defined by an upper and lower bound.

This method of encoding has the advantage of being easy to understand and makes no assumptions as to the structure of the problem. Any conditional parameters (parameters which rely on another parameter(s) for activation) are generated as normal but resolved (removed) by way of a post-processing step.

6.2.4 Population

Typically GAs will adopt one of two approaches for maintaining the population: steady state (incremental) GA, and generational GA. The former replaces some fraction of the population in every generation while the later replaces the entire population every generation. ReACTR uses a standard GA problem representation where the configuration pool represents the population with each configuration in the pool representing a different individual or chromosome. Each parameter in a configuration is a single gene. Crossover and mutation, covered more extensively in the next section, are largely handled in the same way as traditional GAs.

6.2.5 Tournaments and Fitness Function

As runs of the optimisation algorithms that ReACTR is designed to configure are typically expensive, it is imperative that we keep the number of function evaluations to a minimum. We must also gather enough information in order to effectively rank and identify promising configurations. These conflicting requirements mean we must deviate slightly from traditional GA tournaments which typically run all or most of the population at each generation. GGA avoids this problem by dividing its population into two distinct groups, one competitive and one non-competitive.

We instead opt for a different approach by using a global ranking and leader board. The tournaments themselves consist of races with a small number of configurations, the results of which are then aggregated into the overall leader board using a ranking system. Typically, the results of these races are heavily censored due to the early termination of all other runs once a solution has been determined. Given this limitation, and the fact the fact that the algorithms being configured are typically not deterministic [HO15], we require a robust ranking system to properly determine the merit of each configuration.

In ReACTR, we opt to use the TrueSkill ranking system to compute the global leader board. As outlined previously, TrueSkill has a number of properties that make it ideal for this purpose, namely:

- TrueSkill is a Bayesian skill ranking system so naturally handle uncertainty in its ratings [HMG06].

- The system converges to a stable ranking far quicker than other rating systems. The paper introducing TrueSkill claims "*TrueSkill comes close to the information theoretic limit of* $nlog(n)$ *bits to encode a ranking of* n *players.*"
- TrueSkill is used to rank hundreds of thousands of players in online games so it is also very computationally efficient.

These attributes allow us to establish an accurate global ranking of the configurations quickly using only the censored tournament ranking data.

While this approach achieves our real-time configuration goals, that is not to say the approach is without its drawbacks, primary amongst which is ReACTR's inability to account for problem difficulty. Typically offline algorithms aim to optimise a penalised average runtime (PAR) score, usually PAR10 (where timeouts count as as ten times the cutoff value). In an offline configuration scenario, the same problem instance is solved multiple times by different configurations. PAR10 serves the purpose of allowing direct comparison between runtime distributions while also penalising configurations which fail to solve instance within the budget. Without such a penalisation, configurations that find a solution near the end of the allotted time would be considered approximately equal to configurations which timeout e.g. 295s SOLVE $\approx$ 300s TIMEOUT. Distinguishing between these scenarios is particularly important for configurators which adopt a model-based approach to steer the search [HHL11, AMS$^+$15].

Recent work has highlighted flaws with the penalised average runtime methodology, namely that the magnitude of the penalty factor produces a bias in favour of reducing the failure rate [BT18, BKT20]. It instead suggests that the problem is more naturally framed as a multi-objective problem and that it is beneficial to use the commonly used multi-objective performance metric hypervolume (in order to balance both runtime and failure reduction).

In our ReACTR work, we focus solely on the former objective, specifically reducing the *median* solving time. This is not simply a design choice, but a core property of the way that the system works. ReACTR, by design, solves every problem instance once and once only. This methodology, rooted in the practical application of the configurator, means that a direct comparison between configurations on the same problem instance is not possible. Instead ReACTR uses result ranking and so does not require that failed runs be penalised, only that the relevant ranking is correct. The solving time is not considered, nor could it be; when an instance is solved (or fails to be solved) only once we cannot infer whether the solving time is a result of instance difficulty, configuration quality, or some combination of both.

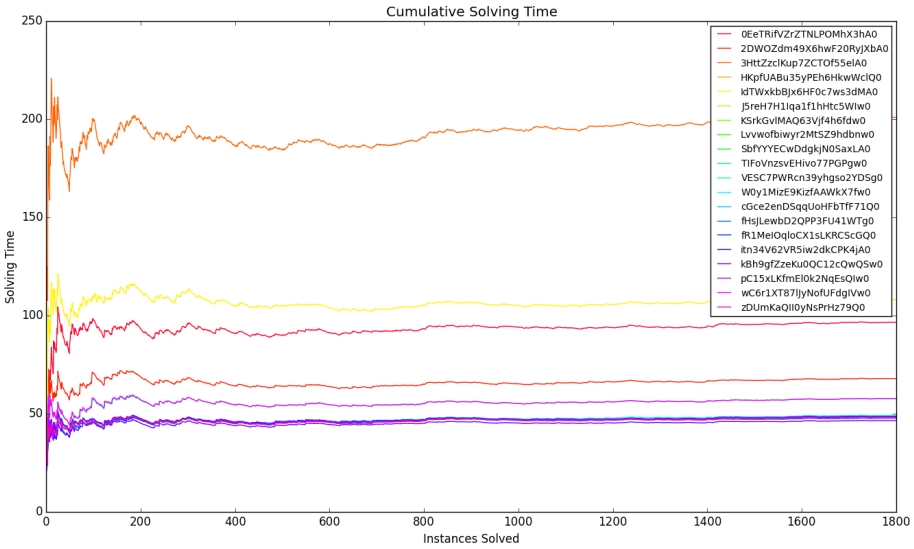
Despite this limitation we show that using TrueSkill as a fitness function, by providing an aggregate score based on the rankings, correlates well with the ground truth solving times. Figures 6.5a and 6.5b show the cumulative average solving time and TrueSkill score, respectively, for twenty configurations selected from previous ReACTR configuration runs on combinatorial auctions solved using CPLEX. These configurations were run to completion and the TrueSkill score was computed by randomly sampling six configurations for each instance and simulating a race.

While ReACTR's design precludes us from allowing multiple configurations to finish, it is still possible to alter the configurator parameters slightly in order to make the ranking more robust without out incurring much overhead. Specifically, we explore two possible choices: changing minimum solving time needed before a race is considered eligible for ranking, and allowing a short slack period after a solution is found during which configurations are treated as equally ranked. The former, shown in Figure 6.6a, is designed to reduce the influence of easy instances on the overall configuration ranking (though we risk inadvertently ignoring fast solutions due to good configurations). The idea behind the latter, displayed in Figure 6.6b, is that configurations that perform approximately equally will be ranked as such (however this comes at the cost of a marginal increase in runtime). Comparing TrueSkill ranking using these adjustments to the ground truth ranking by cumulative average solving time in Figure 6.5a we see that correlation is largely unaffected.

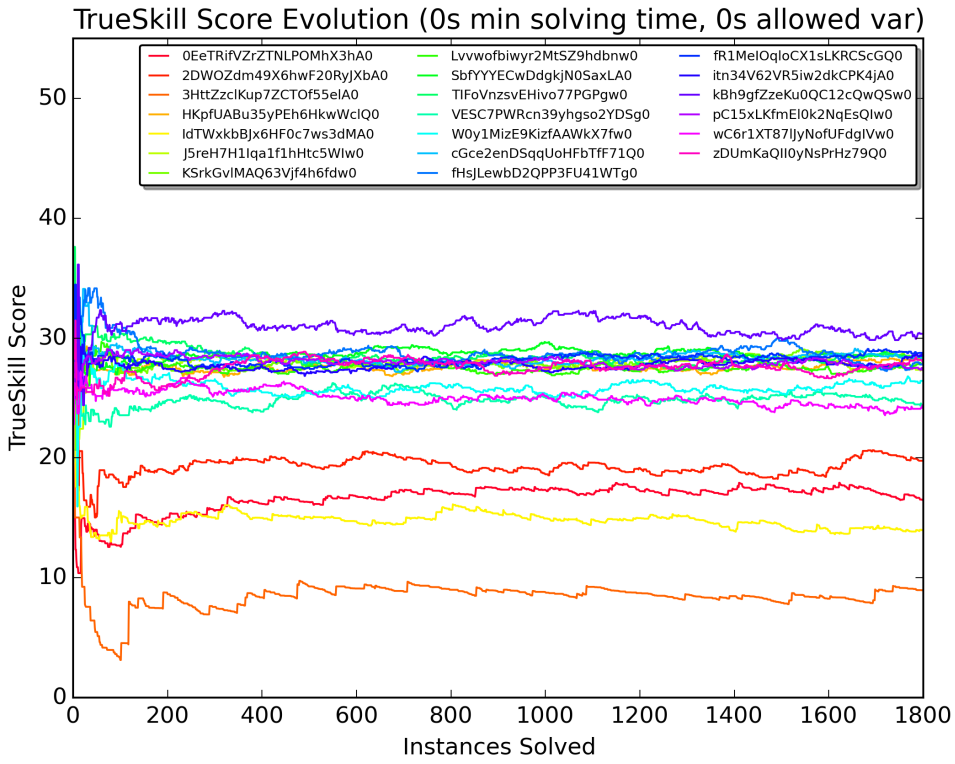
6.2.5.1 Parent Selection

Despite the somewhat unconventional population model and bi-level fitness function, the end result is the same, a complete ranking of the population. This allows us to select the fittest parents to reproduce and recombine to produce potentially even fitter offspring. There are a number of techniques for doing this with some of the more popular being roulette-wheel selection, stochastic universal sampling, truncated selection and tournament selection. There are also numerous other selection methods and adaptations of these methods but in the interest of brevity we will focus our attention only on those which have successfully been used for algorithm configuration.

Tournament selection is a popular method for parent selection, in fact a variant of this is used in the GA based algorithm configurators GGA and GGA++. This approach chooses subgroups of individuals then selects the fittest individuals from within these subgroups as the parents. Tournament selection is not currently implemented in the ReACTR configurator, though there is no technical reason that it can not be implemented as part of the framework in the future.

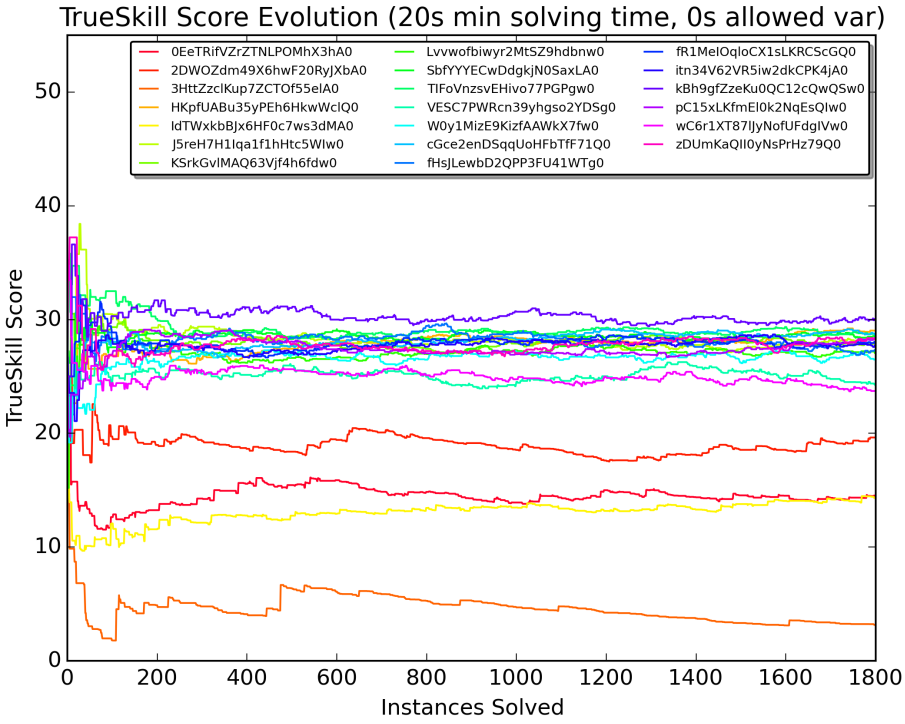


(a) Cumulative average solving times for twenty distinct configurations.

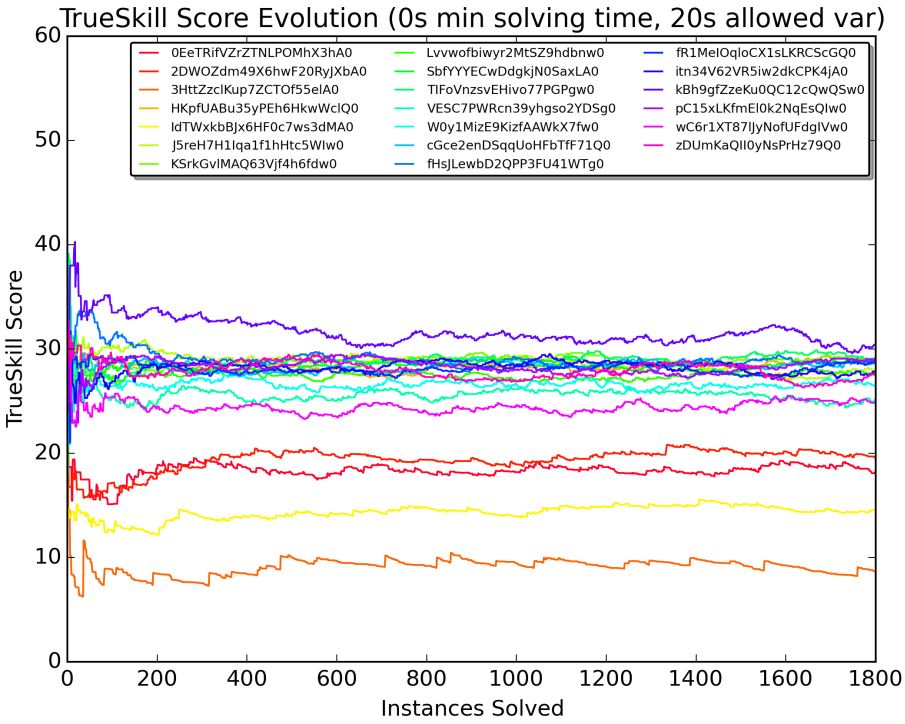


(b) Trueskill ranking of configurations from figure 6.5a (no minimum solving time, no solution slack time).

Figure 6.5: The cumulative average solving time compared to TrueSkill score.



(a) Trueskill ranking of configurations from figure 6.5a (20s minimum solving time, no solution slack time).



(b) Trueskill ranking of configurations from figure 6.5a (no minimum solving time, 20s solution slack time).

Figure 6.6: TrueSkill score under various adjustments of ranking procedure.

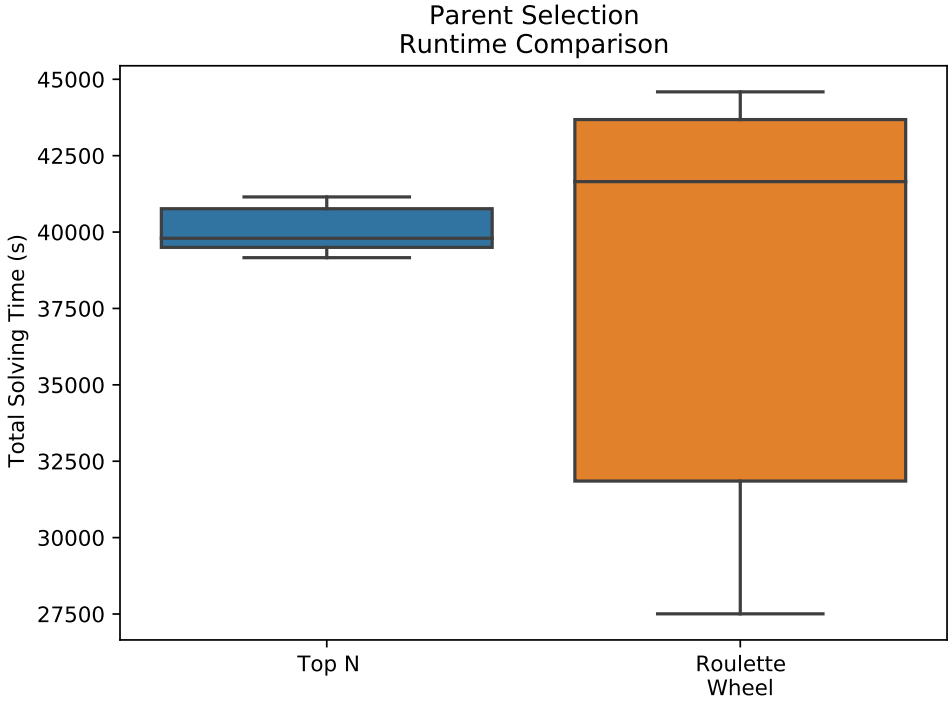


Figure 6.7: Roulette wheel vs. Top *n* parent selection.

Turning our attention to the selection methods that ReACTR offers, Truncated selection selects parents for breeding from only a percentage of the fittest chromosomes. This method ensures that the GA procedure breeds promising individuals but can also result in a lack diversity in the pool.

Roulette-wheel selection and stochastic universal sampling behave similarly in that both scale an individuals probability of selection relative to its fitness function. To use the roulette wheel analogy, the size of the wheel segments are proportional to the individuals fitness and so it is more likely the ball will land on fitter individuals. Where these methods differ is that roulette-wheel selection performs multiple independent spins while stochastic universal sampling performs a single spin but selects multiple points.

While our original ReACTR work uses truncated selection, specifically the top five configurations in the leaderboard, this approach risks causing the pool to stagnate due to lack of diversity. The boxplot in figure 6.7 shows the distribution of solving times for six runs each of truncated selection (top 5 by TrueSkill score) and roulette wheel selection. The experiment is run using Lingeling on the Circuit Fuzz dataset outlined in Section 4.3.4.2.

The first thing we notice is the difference in variance between the two parent selection

methods. The Top-N runtimes are relatively homogeneous with a lower median solving time whereas using roulette wheel selection leads to more variance in running time. This result is in keeping with our understanding and intuition of these selection methods. Roulette wheel selection will explore more diverse, potentially slower, configurations which can impact the solving time. However, the trade-off is the ability to find new and potentially far better configurations resulting in the superior performance for a number of runs. Top N on the other hand will actively exploit good configurations found early on leading to more stable solving times at the expense of stagnation.

6.2.6 Crossover

Crossover is a genetic operator that produces children by combining genes from multiple parents. There are a plethora of different options to choose from when performing crossover. To start with, one must decide how many parents to select for breeding purposes. Typically two parents are selected but some methods also use three or more parents. After deciding how many parents to use the next question is to select the crossover method. The three most common choices for this are uniform, single-point, and multi-point crossover. Uniform crossover treats each gene in the chromosome individually and selects the child value with a certain probability from either parent. Single-point crossover on the other hand selects a single point in the chromosome and passes on all genes prior to that point from one parent and everything after from the other parent. Multi-point crossover is just a generalisation of this approach that selects multiple crossover points instead of one. There are also more exotic mechanisms which rely or preserve certain properties of the problem but in the interest of brevity we will avoid covering these. In ReACTR we use uniform crossover as this makes no assumptions about configuration structure and allows for a larger variety of offspring. With that said, there is no technical reason that single or multi-point crossover can not be used within the ReACT framework.

6.2.7 Mutation

In order to introduce more diversity GAs often randomly mutate a small portion of the chromosomes genes. This is done by allowing each child configuration to select, with some small probability, another value from its valid values instead of adopting one of its parent's values. These values are selected uniformly at random similar to the randomisation procedure described in section 6.2.2. In ReACTR the default probability of mutation is set at 0.05.

6.3 Model-Based Configuration Space Reduction

Algorithm configurators typically fall into two categories: model-based and stochastic local search (as well as a number of hybrid approaches). While the bulk of our configuration generation work, thus far, has focused on evolutionary algorithms (a type of stochastic local search), we now turn our attention to model-based approaches. We do this in order to demonstrate the flexibility of the ReACT framework, offer discussion around what may be possible, and to lay the groundwork for potential avenues of future research.

6.3.1 Motivation

Specifically, we investigate supervised learning approaches, both regression and classification, with the aim of increasing the efficiency of ReACTR's search procedure. We approach in two ways: by training models to predict the quality of individual configurations, and by using the trained model's estimation of feature importance to reduce the configuration space.

The former has been well studied in the context of offline algorithm configuration [HHL11, AMS+15]. Here, we investigate if similar approaches can be applied in an online context where the trade-off between training and solving time is more evident.

Often the most challenging aspect of algorithm configuration is the sheer size of the configuration space to be sampled and explored. Advanced solvers such as CPLEX and Lingeling expose a large number of parameters to the end user. While this is an advantage in that it makes the solvers highly flexible and tunable, this flexibility comes with the cost of a greatly increased configuration space. This configuration space grows exponentially with the number of parameters options exposed. Lingeling offers 241 tunable parameters leading to $\approx 1.1 \times 10^{136}$ potential configurations while CPLEX exposes 74 parameters and has a configuration space size of $\approx 2.3 \times 10^{46}$. In fact, the calculations presented here are optimistic estimates in that they are based on a discretised version of the configuration space offered by ACLib, in reality the space is infinite due to continuous parameters. With search spaces so vast that it is impossible to sample any more than a tiny fraction in any sort of reasonable time. Luckily for us, just as the addition of parameters leads to a combinatorial explosion in options, their removal results in an exponential reduction.

There is mounting evidence that in many cases only a handful of parameters are responsible for the bulk of the solver performance improvement [HHL13, HHL14, FH16]. Despite this, the majority work that we are aware of in this area focuses

on post hoc analysis to identify important parameters or tools to visualise parameter impact [FLH15, BMLH18]. These require that a configuration run has already taken place in order to collect the necessary data. This configuration data is then used to identify promising parameters using forward selection, functional ANOVA, ablation analysis, or some other technique. These approaches are typically very costly both in terms of the initial data acquisition and additional runs required for analysis. Techniques such as surrogate models have been shown to dramatically improve performance but still require time to collect training data and to train the surrogate model [BLE+17].

We are aware of only one configurator, *Golden Parameter Search* (GPS), that actively seeks to exploit this property as part of the configuration procedure [PH20]. GPS adopts a brute force approach by configuring each parameter individually in parallel. GPS employes a bandit approach to determine the order in which to process parameters with those likely to result in improvement getting priority. This method builds on previous configuration landscape work and assumes that parameters are largely independent which allows it to dramatically improve the configuration procedure [PH18].

Here, we propose an alternative method for shrinking the configuration space using off-the-shelf feature selection techniques commonly employed as part of the machine learning pipeline in order to identify parameters which have the largest impact on the configuration performance.

6.3.2 Regression

For our experiments we use random forest regression as our regression model [Bre01]. These models are robust to noise and feature scale while also providing estimates for feature importance. Variants of random forest regression models are used in all current state of the art model-based algorithm configurators [AMS+15, HHL11]. Additionally, random forests have demonstrated considerable success in both surrogate optimisation of hyperparameters, and algorithm portfolios (both closely related to algorithm configuration) [EHHL15, XHHLB12, MSSS13b].

Random forests train multiple decision trees on random subsamples of the training data and features. While individual trees tend to overfit, aggregating the output reduces the variance. Aggregation uses the mean of the trees predictions in the case of regression, and majority vote for classification problems.

Typically configurators train regression models in order to predict the PAR10 score. This model is then used to identify areas where the probability of finding improvement is highest, or to create, assess and filter a list of potential candidate configurations. As

discussed previously, ReACT because of its design is unable to use the PAR metric for direct comparison. Instead, in our regression experiments we aim to predict the TrueSkill score assigned to each configuration.

6.3.3 Classification

While the majority of algorithm configuration work to date has focused on empirical performance models using regression, classification has been largely ignored. This is despite the widespread use and success of classification techniques in algorithm selection [MSSS13a, HKMO14b, AST18].

In this work we explore how we can use a classifier to distinguish good configurations from bad in order to reduce the search space to explore by ReACT. We opt to use a support vector machine (SVM) utilising a linear kernel for our [CV95]. Support Vector Machines are a widely used machine learning technique which are effective in high dimensional spaces such as those encountered in algorithm configuration. They attempt to find a hyperplane which maximises the separation between data points of different classes.

Classification techniques are also particularly well-suited to our goal of configuration space reduction in that we wish to identify all features (parameters) that are contributors to a certain class (high performance configurations). For this reason we train our classifier to distinguish between configurations which have won a race previously and those that have not. Superficially this may seem like an ill-defined class target, as race winners are dependent on the competition they encounter, rather than any inherent trait. However, as ReACT always includes the incumbent configuration, we know that any configuration that wins its race is likely to have set one or more parameters which provide a benefit to the configuration procedure (at least on the problem instance solved.) Our ultimate goal is not classification, but to determine which parameters are strong indicators of success so that we can focus our search efforts on these. We achieve this goal by probing the trained model using feature selection techniques (outlined in detail below) to identify parameters which are the strongest predictors of success or failure.

6.3.4 Feature Selection

Feature selection is a widely used technique in machine learning to choose the features which provide the best predictive power for a model. In a machine learning context this is desirable in order to speed up training, reduce overfitting, and focus on features

which are most strongly linked to the target class. Here, we exploit this final point to identify parameters correlated with configuration performance. By focusing on a subset of important parameters we can reduce the size of the configuration space dramatically.

Feature selection techniques can broadly be divided into two categories: *minimal-optimal* and *all-relevant* [NPBT07]. The former aims to reduce the feature count by eliminating irrelevant and redundant features while maintaining an acceptable predictive performance. The latter attempts to identify all features which are relevant to the prediction of the target class, redundant features are not removed. *All-relevant* feature selection has been shown to be much harder than *minimal-optimal*. For the configuration space reduction scenario that we propose *all-relevant* selection is more suitable but has been proven to be a much harder problem [NPBT07]. Because of this we investigate both types using an *all-relevant* technique, Boruta, in offline tests to demonstrate the viability of this approach and using Random Forests built-in variable importance metric in our online experiments where speed is an important consideration.

Boruta is a widely used *all-relevant* feature selection method [KR$^+$10]. It identifies relevant features by adding shuffled copies of each variable, called shadow variables, to the training set. A random forest classifier is then trained on all of the variables (real and shadow). This model provides Z scores for all features with the maximum Z score amongst the shadow variables set as a threshold. Every feature which scores above this threshold is considered a hit. A statistical test is used to determine the importance of the undecided features. Those that score significantly higher than the threshold are considered important while those which are significantly lower are labelled unimportant. This procedure works in an iterative manner with different shadow variables generated at each iteration until all features have been assigned a label. For these experiments we use the widely used *BorutaPy* Python implementation of this algorithm.

For online scenarios we use Scikit learn's "SelectFromModel" function which in turn takes its feature importances from the random forest's built in calculation of importance [PVG$^+$11]. This is a *minimal-optimal* feature reduction technique that we chose primarily for its speed. In our experiments we use the Scikit's default *Gini Impurity* as the metric to split decision trees on [BFOS84]. This score is then weighted by number of samples routed to that node (an estimate of the probability of reaching the node) and then averaged across all trees in the random forest. As the *Gini Impurity* is calculated as part of constructing the random forest the additional work required to compute the feature importances is low.

6.3.5 Experimental Setup

To test our hypothesis that feature selection can be used to reduce the configuration space and improve configurator performance we devise two experiments: one offline and one online. For the offline experiment we use previously collected algorithm configuration ReACTR run data to train our model and perform feature selection prior to running ReACTR online on the reduced configuration space. The aim here is not to develop an approach which can be integrated into ReACTR, rather to evaluate if feature selection has any merit as a method for computing parameter importance, disregarding computational cost. The online experiments then extends this idea in such a way as to be of practical benefit to the ReACT configuration procedure. Here the reduction technique is integrated into the configuration loop and used during the generation of new candidate configurations.

For these experiments we revisit two previous datasets from different domains: we use the shuffled mixture of combinatorial auctions instances solved using CPLEX 12.6 as described in Section 5.2.2 for our offline experiments and online regression experiments. Meanwhile we adopt the Circuit Fuzz dataset solved using Lingeling (as described in Section 4.3.4) to conduct both our online regression and classification trials. All experiments are run using the same hardware and methodology as previously described, namely servers with 2.66Ghz Intel Xeon E5430 processors with 12GB RAM.

The feature vectors are created only from the configuration parameters and do not include any instance features (which are typically used in other model-based approaches). This choice is in keeping with the ReACT design philosophy. The mapping from parameter value to feature value is direct for numerical parameters, while label encoding is applied to categorical parameters. Missing values are imputed with a unique constant value to indicate the absence of a value. Each feature is normalised using standard scaling so that the magnitude of their impact is on the same scale for models where this is relevant.

6.3.6 Offline Experiments

Our offline training data is generated from the results of previous ReACTR configuration runs and uses the TrueSkill score as the target variable. We then use Boruta to identify all features which are relevant to the prediction of this score. Boruta is a wrapper method and so requires that a core model be provided, for this we opt to use random forest regression. All other Boruta parameters are left unchanged from the defaults.

Boruta identifies nine parameters as being relevant to the prediction of the TrueSkill

score. They are described in the CPLEX documentation as follows [IBM14]:

- *mip limits cutsfactor* - Limits the number of cuts that can be added.
- *mip strategy backtrack* - Controls how often backtracking is done during the branching process.
- *mip strategy rinsheur* - Decides how often to apply the relaxation induced neighbourhood search (RINS) heuristic.
- *mip strategy subalgorithm* - Decides which continuous optimizer will be used to solve the subproblems in a MIP, after the initial relaxation.
- *preprocessing aggregator* - Invokes the aggregator to use substitution where possible to reduce the number of rows and columns before the problem is solved.
- *preprocessing symmetry* - Decides whether symmetry breaking reductions will be automatically executed, during the preprocessing phase, in a MIP or LP model.
- *simplex dgradient* - Decides the type of pricing applied in the dual simplex algorithm.
- *simplex pgradient* - Sets the primal simplex pricing algorithm.
- *simplex refactor* - Sets the number of iterations between refactoring of the basis matrix.

Encouragingly, two of the parameters found to be important in this study, *mip limits cutsfactor* and *mip strategy subalgorithm*, are also identified as the most important parameters in previous work on parameter importance using forward selection on the CPLEX BIGMIX dataset [HHL13]. Although this method of assessing parameter importance is still too computationally taxing to be of practical use in an online configuration scenario it is still considerably faster than the majority of alternative methods of identifying parameter importance, with a typical runtime in the order of minutes or hours.

We verify the effectiveness of reducing the configuration space in this manner by configuring CPLEX using both the full configuration space and the reduced configuration space of nine parameters (all other parameter values were fixed at the solver defaults). This reduces the size configuration space from $\approx 2.3 \times 10^{46}$ configurations to just $\approx 9.5 \times 10^{6}$ configurations. The boxplots in Figure 6.8 shows the distribution of total solving times for ten runs of each configuration space. As with previous boxplots the box covers the first to third quartiles with the median denoted by the central line. The whiskers extend beyond the first and third quartiles by 1.5 $\times$ the interquartile range.

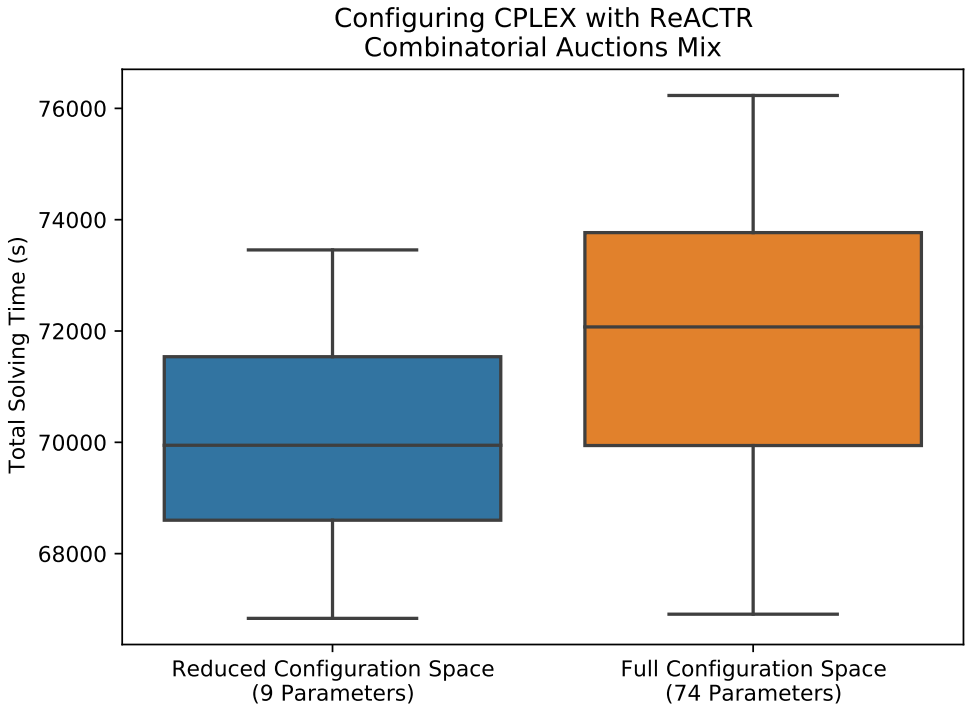


Figure 6.8: CPLEX - Combinatorial Auction Mix: Full vs. Reduced Configuration Space.

We can clearly see that the configuration using reduced configuration space outperforms that with the full configuration space. We ran a one-sided Wilcoxon signed-rank test to confirm that median of the differences can be assumed negative (reduced < full) which gives a confidence level of P<0.01.

6.3.7 Online Experiments

Having demonstrated the effectiveness of configuration space reduction in offline experiments we now examine whether this technique can be of benefit when adapted to work online within ReACT framework. We do this by replacing the expensive Boruta feature selection with SelectFromModel which uses the feature importance computed while building the model. The number of training samples is also limited to a relatively small number examples (in these experiments one hundred) so that the model can be trained quickly. The generation procedure works as follows:

- A highly ranked configuration is selected from the leader-board and copied (here we use roulette wheel selection). This becomes the template for the new configuration.
- SelectFromModel is used to identify the important parameters.

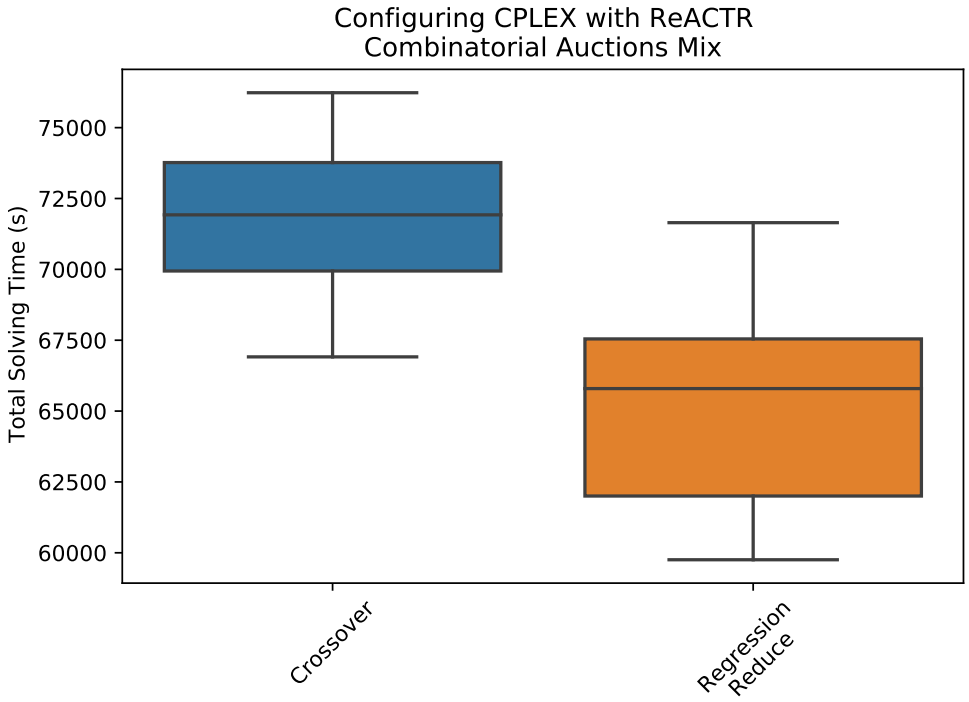


Figure 6.9: CPLEX - Combinatorial Auction Mix: Online Configuration Space Reduction

- Each of the important parameters is then randomly assigned a value from its allowed values.

Figure 6.9 shows the results of the previously conducted offline experiments adapted to the online context. The dataset is the same and a random forest regressor is still tasked with predicting the TrueSkill score. The training set is limited to the results encountered during this training run. Again we can see that reducing the configuration space, this time in a dynamic fashion, results in a large, statistically significant ($P < 0.01$), improvement in performance.

We now examine the same technique applied to a different dataset, circuit fuzzing SAT instances solved using Lingeling. Given Lingeling's configuration space is $\approx 1.1 \times 10^{136}$ this is a challenging task but also the perfect showcase for these reduction techniques. We also take the opportunity to use this test-bed to explore the effectiveness of classification as a reduction technique. Figure 6.10 shows the results of these experiments. TrueSkill score prediction using random forest regressors is again our regression objective while for classification we aim to predict the label "has_won" using a linear support vector classifier.

Reducing the configuration space via regression clearly performs the best, however both techniques perform significantly better than crossover. We confirm this be running

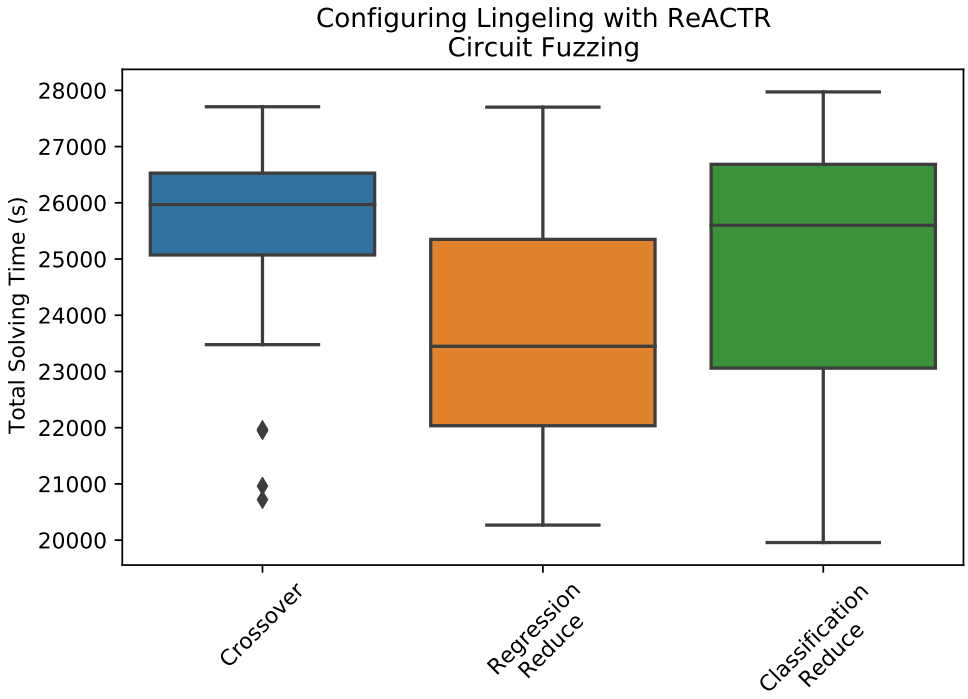


Figure 6.10: Lingeling - Circuit Fuzz: Online Configuration space reduction.

a one-sided Wilcoxon signed-rank test. This shows the regression result to be highly statistically significant ($P \ll 0.001$) while the classification result is significant ($P<0.05$).

6.3.8 Running Time vs. Solution Time Trade-off

It is worth noting that the previous results show only the solver solving time. They do not take into account the time required to train a model and generate the configurations. While running the experiment in Figure 6.10 we tracked the impact of this generation method. As expected the linear support vector classifier method is lightweight, increasing the overall solving time by only 342 (SD=27) seconds on average. The random forest regression model increased the total running time by 755 (SD=43).

Including the overhead of the generation for each run shifts the total solving time distributions. Rerunning the Wilcoxon signed-rank test we find that the classification reduction method is no longer statistically distinguishable from the default crossover strategy. The regression result, though including a more costly training procedure, is still highly statistically significant ($P<0.001$). This result illustrates a key point which resonates throughout this thesis; in real-time algorithm configuration must carefully balance the configuration expense against the solving time benefit.

Though all experiments in this thesis have been run on a fixed set of instances, it is

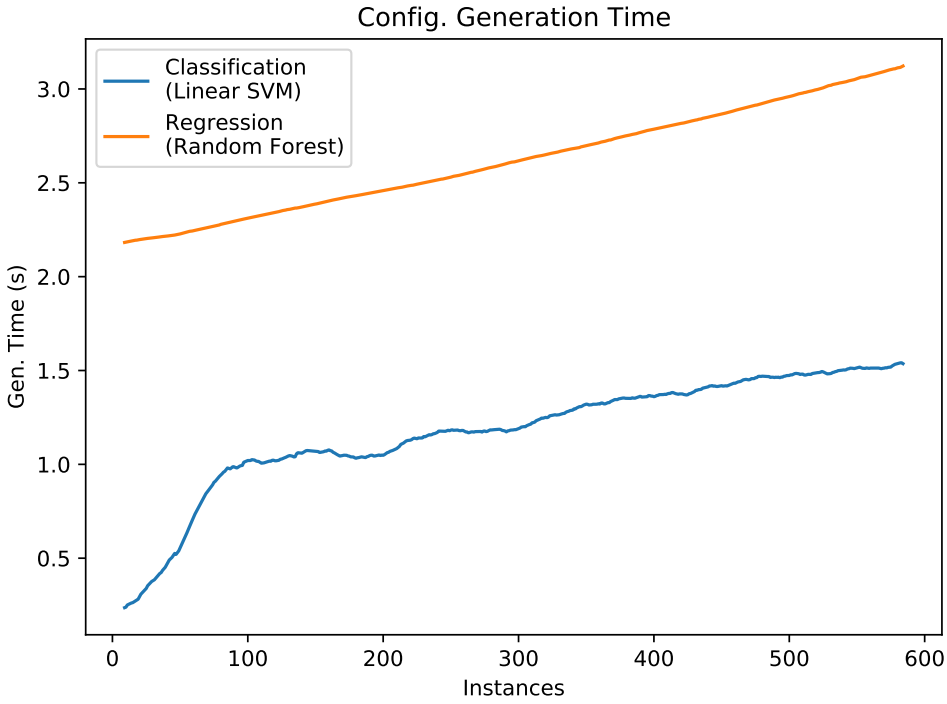


Figure 6.11: Lingeling - Circuit Fuzz: Configuration generation times.

important to remember that the goal is to configure over a potentially infinite stream of instances. As such, objectives must be considered over a time horizon, e.g. in the next 200 instances the investment in configuration will pay off.

Another point worth noting in the context of modelling over streams is that the model training time can increase as the training set increases. Figure 6.11 shows the time taken to train and generate the configurations in Figure 6.10. We can see for both models that training time increases steadily with the number of training instances. This can be avoided using intelligent engineering like subsampling. Efficient new methods such as Mondrian Forests which allow online model updates, or Boruta-like feature reduction techniques using Random Ferns to reduce *all-relevant* feature selection time by orders of magnitude [LRT14, Kur17] also show great promise. The work presented here is only a preliminary exploration of an idea with many potential avenues for investigation.

6.4 Chapter Summary

This chapter looked at a number of metrics available to us which allow us to assess and remove under-performing configurations from the pool. We discuss the pros and cons of each of these approaches, ultimately demonstrating that the robust ranking system TrueSkill is best suited to our needs. We determined how the thresholds chosen for

TrueSkill's skill and confidence score impact the removal and overall success of the configuration procedure.

This chapter also examined how to generate good candidate configurations with a minimum of computational overhead. We outline a novel genetic algorithm procedure, designed to reduce the number of evaluations required by providing an aggregated ranking of competitions across multiple generations. This aggregate ranking then allows us to recombine configurations shown to be high performers using a recombination procedure. We also showed that feature selection techniques from the machine learning literature can be used to reduce the configuration space significantly; in doing so we improve the configuration procedure by focusing the search on parameters with the greatest performance impact.

Chapter 7

Conclusions and Future Work

Summary. *We finish by outlining the conclusions of the work presented and reiterating our contributions to the body of knowledge. We also describe potential avenues of future work which are interesting and worthy of further investigation.*

7.1 Conclusions

We conclude this thesis by outlining a number of novel ideas and contributions introduced within. In this thesis:

- We introduced and motivated the need to study a variant of the algorithm configuration problem, the real-time algorithm configuration problem. This variant of the problem asks *how much configuration can be achieved without impacting the wallclock time needed to return a solution to the end-user?*
- We outlined our proposed framework for solving this problem, the ReACT framework. This framework exploits aggregate ranking, parallel computation and aggressive runtime capping to return solutions as quickly as possible to the user while still inferring enough information to provide a ranking of the candidate configurations. This ranking can then be used to direct the search and achieve constant improvement over a stream of instances.
- We provided two concrete instantiations of the framework and used these to empirically demonstrate that the ReACT framework can achieve performance on a par with, or even exceeding that of current state-of-the-art offline configurators across a variety of solvers and scenarios in the combinatorial optimisation domain.

- We investigated and provided strong experimental evidence and discussion for each of the design choices made, namely the ranking, selection, removal, and generation component instantiations.
- We analysed whether, and to what extent, the ordering of incoming problem instances impacts the performance of our configurator. We also demonstrate that the choice of candidate selection procedure is related to the properties of the instance ordering.
- We exploit the fact that in many configuration scenarios correctly configuring only a handful of parameters is able to achieve the lion's share of performance improvement. We validate this by training models to predict configuration quality then probing these models using off-the-shelf machine learning feature selection techniques to reduce the configuration space to only the parameters which are the greatest indicators of quality.

The thesis defended in this dissertation was:

The performance of combinatorial solvers can be improved as a stream of instances is being processed without prior information or training. This is possible due to real-time algorithm configuration using parallel evaluation combined with a robust ranking system.

We say that the overall conclusion is that it has been defended. We have outlined a framework for improving the configuration of a combinatorial solver while solving a stream of instance thus improving the solver's performance. Additionally we have demonstrated two concrete instantiations of the framework, discussed the methods employed, as well as factors that impact on their performance, and empirically demonstrated their effectiveness.

7.2 Future Work

7.2.1 Configuring the Configurator

In this thesis we have extolled the virtues of algorithm configuration, therefore we would be remiss not to include our own algorithm amongst those that would benefit from automatic algorithm configuration. In this work we have attempted to provide empirical justification for the values adopted by all important components. However, as we have adhered to the *Programming by Optimization* paradigm, many configuration choices have been implemented and exposed to the user [Hoo12a]. Given the size

of the configuration space and the expense of even a single configuration run it is infeasible to conduct a second-order configuration by traditional means. Identifying and applying efficient methods for optimising configurator parameters is an interesting topic of research that we hope to pursue in the future. Two relatively recent methods appearing promising for this:

Surrogate Models Recent work has shown that surrogate models can accurately predict a configuration's performance on an instance in a fraction of the time that it would take to actually solve the instance [EHHL15, ELH+18]. This is achieved by training a model on the results of past configuration runs. Replacing the expensive evaluations performed in the inner loop of the configuration process with low cost surrogate model predictions would bring the cost of configuration to a level where configuring the configurator becomes achievable. Using surrogate models to tune the parameters of ReACTR for maximum general (or even per-set) performance is an area of investigation that is likely to prove fruitful.

Dynamic Configuration We have set fixed parameter values to control various aspects of both concrete instantiations of the ReACT framework. While these static values have been shown to perform well overall, in many cases a value which adapts to the current state of the configuration search would achieve superior performance. The idea of adapting parameters on-the-fly has been studied in the areas of *Reactive Search*, *Evolutionary Algorithms*, and *Dynamic Algorithm Configuration* [BB07, PPdSN19, BBE+20]. Applying some of these ideas to dynamically configure components of the ReACT framework during search would be an interesting to investigate, possibly in combination with the surrogate models outlined above.

7.2.2 Alternative Framework Instantiations and Improvements

The ReACT framework is designed to be modular and extensible. In this work we have presented two possible instantiations of the framework, ReACT and ReACTR, however there are a multitude of potential options for every component of the framework. We are aware of at least one work that has proposed an alternative instantiation of the framework that foregoes the ranking component in favour of using a contextual preselection bandit approach for choosing configurations [MWB+20]. Modelling the problem as a preselection bandit problem allows for better reasoning about the heavily censored run data, while adopting the contextual version of this approach enables configuration to occur on a per-instance basis. This work displays some favourable results in comparison to our ReACTR instantiation however the overhead for the method

used is larger.

We hope that researchers will continue to extend and improve on the foundations laid by this framework. In particular, it would be interesting to port ideas which have been shown to be successful in offline configuration scenarios and adapt them to the real-time configuration context. Another component ripe for improvement is the ranking system adopted by ReACTR. In this thesis we have shown that TrueSkill works well and provides strong results, nevertheless it is still an off-the-shelf ranking system designed for finding balanced games rather than algorithm configuration. A bespoke solution tailored to this problem or a learning to rank approach offer the potential for improvement here.

7.2.3 Exploiting Stochastic Instance Arrivals

Throughout this thesis we have assumed a constant stream of incoming problem instances which we solve sequentially. Though this assumption holds true in many practical scenarios, it is also common to encounter streams where problem instances arrive in at random intervals. ReACT can handle these situations, however in the instantiations that we have presented it does not leverage the properties of the stream distribution to achieve the best performance. We consider improvements for two scenarios: one where a glut of instances has arrived in a short period of time and caused a backlog, and another where there is a drought of instances such that the configurator is left idle.

Instance Backlog When faced with a backlog of problem instances there are a number of strategies that we can adopt. If the backlog is very large it may be preferable to forgo configuration (or reduce the information gained) by distributing the instances amongst the available processors in order to solve the instances quickly, in keeping with the real-time algorithm configuration paradigm. The scale of the backlog could also dictate the exploration and exploitation trade-off in the candidate selection strategy used, for example if there is a large backlog it may be preferable to use more known strong configurations over random configurations.

More advanced techniques such as online scheduling may also improve the processing time. One possible solution from the literature which appears particularly promising for adaptation to the ReACT context is proposed in [DAD18b, DAD18a]. Here the authors develop a machine learning model to predict the expected runtime of an instance which is then used as part of a mixed integer programming model to maximise the number of solved instances. Additionally this work outlines a heuristic-based interruption

procedure to prevent poor predictions impacting the solution process too much. Of particular interest in the ReACT context would be altering the MIP model to increase the configuration speed, for example by processing instances in a particular order as demonstrated in Chapter 5 and in [SHM12, SH13]. We hope to apply these ideas to ReACT as it would improve the configurator's performance in many practical applications where the configurator is likely to be deployed.

Instance Drought On the other extreme there may be periods where the configurator receives no instances to process. For example, if a business only requires solutions during normal working hours then the configurator may be idle at night time and weekends. These periods of downtime could be effectively used to revisit previously solved instances in order to establish things such as the full ranking of a race, the variability in the instance solving time, or how a recently introduced configuration would perform on a historical instance. This blurs the line between real-time and offline algorithm configuration but does not impact on the time needed to return a solution.

Another potential use of a break in instances is to utilise the available time to employ more expensive techniques such as training models or performing analysis over the collected data. Once trained these models or the insights gained can be effectively used to improve the configuration process when instances begin arriving again. The application of these ideas is worthy of investigation as it not only expands the range of techniques available to ReACT but also improves the configurators practical use.

7.2.4 Balancing Configuration Overhead Against Speed-up

In this thesis we have emphasised the need for methods with a low computational overhead due to the real-time nature of the configurator and our desire to return solutions as quickly as possible. In Section 6.3.8 we briefly discussed the impact of training overhead and showed how using a more expensive model can potentially be worthwhile if the time saving it provides exceeds its training cost.

A natural extension of this work is to allow more computationally expensive methods which will "pay for themselves" over a set period of time. More formally, consider two methods (e.g. generation); our current method m_c and another more computationally expensive method m_e. The overhead of using these methods is o_c and o_e respectively. We also define an instance horizon, h, over which we expect the method to break-even or improve the configuration run time. The solving time of the instances in h using method m_e is denoted as t_{eh} (and likewise t_{ch} for solving with the method m_c). We can then say we would prefer the method m_e over m_c iff $o_e + t_{eh} \leq o_c + t_{ch}$. Of course

without an oracle we have no way of knowing the values for t_{eh} and t_{ch} (and to a lesser extent o_e and o_c, though these can likely be estimated by historical data). As such a large part of the challenge here involves estimating these values and adapting the formula to account for the probabilistic nature of these estimations. We suggest this as an inviting area for future research as a solution here would clearly demarcate which methods are applicable to real-time algorithm configuration from those that are best reserved for offline configuration scenarios.

Bibliography

[ACFS02] Peter Auer, Nicolò Cesa-Bianchi, Yoav Freund, and Robert E. Schapire. The nonstochastic multiarmed bandit problem. *SIAM J. Comput.*, 32(1):48–77, 2002.

[AGM14] Roberto Amadini, Maurizio Gabbrielli, and Jacopo Mauro. SUNNY: a lazy portfolio approach for constraint solving. *Theory Pract. Log. Program.*, 14(4-5):509–524, 2014.

[AGMS16] Carlos Ansótegui, Joel Gabàs, Yuri Malitsky, and Meinolf Sellmann. Maxsat by improved instance-specific algorithm configuration. *Artif. Intell.*, 235:26–39, 2016.

[AHS10] Alejandro Arbelaez, Youssef Hamadi, and Michèle Sebag. Continuous search in constraint programming. In *22nd IEEE International Conference on Tools with Artificial Intelligence, ICTAI 2010, Arras, France, 27-29 October 2010 - Volume 1*, pages 53–60. IEEE Computer Society, 2010.

[AHS12] Alejandro Arbelaez, Youssef Hamadi, and Michèle Sebag. Continuous search in constraint programming. In Youssef Hamadi, Eric Monfroy, and Frédéric Saubion, editors, *Autonomous Search*, pages 219–243. Springer, 2012.

[AKM05] Tobias Achterberg, Thorsten Koch, and Alexander Martin. Branching rules revisited. *Oper. Res. Lett.*, 33(1):42–54, 2005.

[AL06] Belarmino Adenso-Díaz and Manuel Laguna. Fine-tuning of algorithms using fractional experimental designs and local search. *Oper. Res.*, 54(1):99–114, 2006.

[AMS+15] Carlos Ansótegui, Yuri Malitsky, Horst Samulowitz, Meinolf Sellmann, and Kevin Tierney. Model-based genetic algorithms for algorithm configuration. In *IJCAI*, pages 733–739, 2015.

[AO06] Charles Audet and Dominique Orban. Finding optimal algorithmic parameters using derivative-free optimization. *SIAM J. Optim.*, 17(3):642–664, 2006.

[AS09] Gilles Audemard and Laurent Simon. Glucose: a solver that predicts learnt clauses quality. *SAT Competition*, pages 7–8, 2009.

[AST09a] C. Ansotegui, M. Sellmann, and K. Tierney. A gender-based genetic algorithm for the automatic configuration of algorithms. *CP*, pages 142–157, 2009.

[AST09b] Carlos Ansótegui, Meinolf Sellmann, and Kevin Tierney. A gender-based genetic algorithm for the automatic configuration of algorithms. In *Principles and Practice of Constraint Programming-CP 2009*, pages 142–157. Springer, 2009.

[AST18] Carlos Ansótegui, Meinolf Sellmann, and Kevin Tierney. Self-configuring cost-sensitive hierarchical clustering with recourse. In John N. Hooker, editor, *Principles and Practice of Constraint Programming - 24th International Conference, CP 2018, Lille, France, August 27-31, 2018, Proceedings*, volume 11008 of *Lecture Notes in Computer Science*, pages 524–534. Springer, 2018.

[ATC13] Alejandro Arbelaez, Charlotte Truchet, and Philippe Codognet. Using sequential runtime distributions for the parallel speedup prediction of SAT local search. *Theory Pract. Log. Program.*, 13(4-5):625–639, 2013.

[ATO16] Alejandro Arbelaez, Charlotte Truchet, and Barry O'Sullivan. Learning sequential and parallel runtime distributions for randomized algorithms. In *28th IEEE International Conference on Tools with Artificial Intelligence, ICTAI 2016, San Jose, CA, USA, November 6-8, 2016*, pages 655–662. IEEE Computer Society, 2016.

[ATY00] Arne Andersson, Mattias Tenhunen, and Fredrik Ygge. Integer programming for combinatorial auction winner determination. In *MultiAgent Systems, 2000. Proceedings. Fourth International Conference on*, pages 39–46. IEEE, 2000.

[Bar99] Roman Barták. Constraint programming: In pursuit of the holy grail. In *Proceedings of the Week of Doctoral Students (WDS99)*, volume 4, pages 555–564. MatFyzPress Prague, 1999.

[BB07] Mauro Brunato and Roberto Battiti. Reactive search. In Teofilo F. Gonza-

lez, editor, *Handbook of Approximation Algorithms and Metaheuristics*. Chapman and Hall/CRC, 2007.

[BB12] James Bergstra and Yoshua Bengio. Random search for hyper-parameter optimization. *Journal of Machine Learning Research*, 13(Feb):281–305, 2012.

[BBBK11] James Bergstra, Rémi Bardenet, Yoshua Bengio, and Balázs Kégl. Algorithms for hyper-parameter optimization. In John Shawe-Taylor, Richard S. Zemel, Peter L. Bartlett, Fernando C. N. Pereira, and Kilian Q. Weinberger, editors, *Advances in Neural Information Processing Systems 24: 25th Annual Conference on Neural Information Processing Systems 2011. Proceedings of a meeting held 12-14 December 2011, Granada, Spain*, pages 2546–2554, 2011.

[BBD+12] A. Balint, A. Belov, D. Diepold, S. Gerber, M. Järvisalo, and C. Sinz, editors. *Proceedings of SAT Challenge 2012*, 2012.

[BBE+20] André Biedenkapp, H. Furkan Bozkurt, Theresa Eimer, Frank Hutter, and Marius Lindauer. Dynamic algorithm configuration: Foundation of a new meta-algorithmic framework. In Giuseppe De Giacomo, Alejandro Catalá, Bistra Dilkina, Michela Milano, Senén Barro, Alberto Bugarín, and Jérôme Lang, editors, *ECAI 2020 - 24th European Conference on Artificial Intelligence, 29 August-8 September 2020, Santiago de Compostela, Spain, August 29 - September 8, 2020 - Including 10th Conference on Prestigious Applications of Artificial Intelligence (PAIS 2020)*, volume 325 of *Frontiers in Artificial Intelligence and Applications*, pages 427–434. IOS Press, 2020.

[BBH+13] Adrian Balint, Anton Belov, Marijn JH Heule, Matti Järvisalo, et al. Proceedings of sat competition 2013. University of Helsinki, 2013.

[BBLP05] Thomas Bartz-Beielstein, Christian WG Lasarczyk, and Mike Preuß. Sequential parameter optimization. In *2005 IEEE congress on evolutionary computation*, volume 1, pages 773–780. IEEE, 2005.

[BBS07] Prasanna Balaprakash, Mauro Birattari, and Thomas Stützle. Improvement strategies for the f-race algorithm: Sampling design and iterative refinement. In *International workshop on hybrid metaheuristics*, pages 108–122. Springer, 2007.

[BBvB17] R. Burger, M. Bharatheesha, M. van Eert, and R. Babuška. Automated

tuning and configuration of path planning algorithms. In *2017 IEEE International Conference on Robotics and Automation (ICRA)*, pages 4371–4376, 2017.

[BC12] Sébastien Bubeck and Nicolò Cesa-Bianchi. Regret analysis of stochastic and nonstochastic multi-armed bandit problems. *Found. Trends Mach. Learn.*, 5(1):1–122, 2012.

[BCC93] Egon Balas, Sebastián Ceria, and Gérard Cornuéjols. A lift-and-project cutting plane algorithm for mixed 0–1 programs. *Mathematical programming*, 58(1-3):295–324, 1993.

[BCR12] Nicolas Beldiceanu, Mats Carlsson, and Jean-Xavier Rampon. Global constraint catalog, (revision a), 2012.

[BDHJ14] Anton Belov, Daniel Diepold, Marijn JH Heule, and Matti Järvisalo. Proceedings of sat competition 2014. University of Helsinki, 2014.

[Bes06] Christian Bessiere. Chapter 3 - constraint propagation. In Francesca Rossi, Peter van Beek, and Toby Walsh, editors, *Handbook of Constraint Programming*, volume 2 of *Foundations of Artificial Intelligence*, pages 29 – 83. Elsevier, 2006.

[BFOS84] Leo Breiman, J. H. Friedman, R. A. Olshen, and C. J. Stone. *Classification and Regression Trees*. Wadsworth, 1984.

[BG17] Martin Bichler and Jacob K Goeree. *Handbook of spectrum auction design*. Cambridge University Press, 2017.

[BGG+71] Michel Bénichou, Jean-Michel Gauthier, P. Girodet, Gerard Hentges, Gerard Ribière, and O. Vincent. Experiments in mixed-integer linear programming. *Math. Program.*, 1(1):76–94, 1971.

[BH+16] Tomáš Balyo, Marijn JH Heule, et al. Proceedings of sat competition 2016. University of Helsinki, 2016.

[BHJ17] Tomáš Balyo, Marijn JH Heule, and Matti Järvisalo. Proceedings of sat competition 2017: Solver and benchmark descriptions. University of Helsinki, Department of Computer Science, 2017.

[BHLS04] Frédéric Boussemart, Fred Hemery, Christophe Lecoutre, and Lakhdar Sais. Boosting systematic search by weighting constraints. In *ECAI*, volume 16, page 146, 2004.

[BHvMW09] Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh. Conflict-driven clause learning sat solvers. *Handbook of Satisfiability, Frontiers in Artificial Intelligence and Applications*, pages 131–153, 2009.

[Bie10] Armin Biere. Lingeling, plingeling, picosat and precosat at sat race 2010. *FMV Report Series Technical Report*, 10(1), 2010.

[Bie13] Armin Biere. Lingeling, plingeling and treengeling entering the sat competition 2013. *Proceedings of SAT competition*, 2013:1, 2013.

[Bir05] Mauro Birattari. *The problem of tuning metaheuristics: as seen from the machine learning perspective*. PhD thesis, Brussels, Univ., 2005.

[BJS97] Roberto J Bayardo Jr and Robert Schrag. Using csp look-back techniques to solve real-world sat instances. In *AAAI/IAAI*, pages 203–208, 1997.

[BKK+16] Bernd Bischl, Pascal Kerschke, Lars Kotthoff, Marius Lindauer, Yuri Malitsky, Alexandre Fréchette, Holger Hoos, Frank Hutter, Kevin Leyton-Brown, Kevin Tierney, et al. Aslib: A benchmark library for algorithm selection. *Artificial Intelligence*, 237:41–58, 2016.

[BKT20] Jakob Bossek, Pascal Kerschke, and Heike Trautmann. A multi-objective perspective on performance assessment and automated selection of single-objective optimization algorithms. *Appl. Soft Comput.*, 88:105901, 2020.

[BLB10] Robert Brummayer, Florian Lonsing, and Armin Biere. Automated testing and debugging of SAT and QBF solvers. In Ofer Strichman and Stefan Szeider, editors, *Theory and Applications of Satisfiability Testing - SAT 2010, 13th International Conference, SAT 2010, Edinburgh, UK, July 11-14, 2010. Proceedings*, volume 6175 of *Lecture Notes in Computer Science*, pages 44–57. Springer, 2010.

[BLCW09] Yoshua Bengio, Jérôme Louradour, Ronan Collobert, and Jason Weston. Curriculum learning. In Andrea Pohoreckyj Danyluk, Léon Bottou, and Michael L. Littman, editors, *Proceedings of the 26th Annual International Conference on Machine Learning, ICML 2009, Montreal, Quebec, Canada, June 14-18, 2009*, volume 382 of *ACM International Conference Proceeding Series*, pages 41–48. ACM, 2009.

[BLE+17] Andre Biedenkapp, Marius Lindauer, Katharina Eggensperger, Frank Hutter, Chris Fawcett, and Holger H. Hoos. Efficient parameter importance analysis via ablation with surrogates. In Satinder P. Singh and Shaul

Markovitch, editors, *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence, February 4-9, 2017, San Francisco, California, USA*, pages 773–779. AAAI Press, 2017.

[BLP20] Yoshua Bengio, Andrea Lodi, and Antoine Prouvost. Machine learning for combinatorial optimization: A methodological tour d'horizon. *European Journal of Operational Research*, 2020.

[BM04] Thomas Bartz-Beielstein and Sandor Markon. Tuning search algorithms for real-world applications: a regression tree based approach. In *Proceedings of the IEEE Congress on Evolutionary Computation, CEC 2004, 19-23 June 2004, Portland, OR, USA*, pages 1111–1118. IEEE, 2004.

[BMLH18] Andre Biedenkapp, Joshua Marben, Marius Lindauer, and Frank Hutter. CAVE: configuration assessment, visualization and evaluation. In Roberto Battiti, Mauro Brunato, Ilias S. Kotsireas, and Panos M. Pardalos, editors, *Learning and Intelligent Optimization - 12th International Conference, LION 12, Kalamata, Greece, June 10-15, 2018, Revised Selected Papers*, volume 11353 of *Lecture Notes in Computer Science*, pages 115–130. Springer, 2018.

[BR96] Christian Bessiere and Jean-Charles Régin. Mac and combined heuristics: Two reasons to forsake fc (and cbj?) on hard problems. In *International Conference on Principles and Practice of Constraint Programming*, pages 61–75. Springer, 1996.

[Bra53] Ralph Allan Bradley. Some statistical methods in taste testing and quality evaluation. *Biometrics*, 9(1):22–38, 1953.

[Bré79] Daniel Brélaz. New methods to color the vertices of a graph. *Communications of the ACM*, 22(4):251–256, 1979.

[Bre01] Leo Breiman. Random forests. *Mach. Learn.*, 45(1):5–32, 2001.

[BS12] Adrian Balint and Uwe Schöning. Choosing probability distributions for stochastic local search and the role of make versus break. In Alessandro Cimatti and Roberto Sebastiani, editors, *Theory and Applications of Satisfiability Testing - SAT 2012 - 15th International Conference, Trento, Italy, June 17-20, 2012. Proceedings*, volume 7317 of *Lecture Notes in Computer Science*, pages 16–29. Springer, 2012.

[BSPV02] Mauro Birattari, Thomas Stützle, Luís Paquete, and Klaus Varrentrapp. A racing algorithm for configuring metaheuristics. In William B. Langdon,

Erick Cantú-Paz, Keith E. Mathias, Rajkumar Roy, David Davis, Riccardo Poli, Karthik Balakrishnan, Vasant G. Honavar, Günter Rudolph, Joachim Wegener, Larry Bull, Mitchell A. Potter, Alan C. Schultz, Julian F. Miller, Edmund K. Burke, and Natasa Jonoska, editors, *GECCO 2002: Proceedings of the Genetic and Evolutionary Computation Conference, New York, USA, 9-13 July 2002*, pages 11–18. Morgan Kaufmann, 2002.

[BT52] Ralph Allan Bradley and Milton E Terry. Rank analysis of incomplete block designs: I. the method of paired comparisons. *Biometrika*, 39(3/4):324–345, 1952.

[BT94] R. Battiti and G. Tecchiolli. The reactive tabu search. *ORSA Journal on Computing*, 6(2):126–140, 1994.

[BT18] Jakob Bossek and Heike Trautmann. Multi-objective performance measurement: Alternatives to PAR10 and expected running time. In Roberto Battiti, Mauro Brunato, Ilias S. Kotsireas, and Panos M. Pardalos, editors, *Learning and Intelligent Optimization - 12th International Conference, LION 12, Kalamata, Greece, June 10-15, 2018, Revised Selected Papers*, volume 11353 of *Lecture Notes in Computer Science*, pages 215–219. Springer, 2018.

[BTH14] Sam Bayless, Dave A. D. Tompkins, and Holger H. Hoos. Evaluating instance generators by configuration. In Panos M. Pardalos, Mauricio G. C. Resende, Chrysafis Vogiatzis, and Jose L. Walteros, editors, *Learning and Intelligent Optimization - 8th International Conference, Lion 8, Gainesville, FL, USA, February 16-21, 2014. Revised Selected Papers*, volume 8426 of *Lecture Notes in Computer Science*, pages 47–61. Springer, 2014.

[BY13] Joseph Bebel and Henry Yuen. Hard sat instances based on factoring. *SAT Competition*, page 102, 2013.

[BYBS10] Mauro Birattari, Zhi Yuan, Prasanna Balaprakash, and Thomas Stützle. F-race and iterated f-race: An overview. In *Experimental methods for the analysis of optimization algorithms*, pages 311–336. Springer, 2010.

[CAG+14] José Cáceres-Cruz, Pol Arias, Daniel Guimarans, Daniel Riera, and Angel A. Juan. Rich vehicle routing problem: Survey. *ACM Comput. Surv.*, 47(2):32:1–32:28, 2014.

[CD09] Imran Ali Chaudhry and Paul R Drake. Minimizing total tardiness for the machine scheduling and worker assignment problems in identical parallel machines using genetic algorithms. *The International Journal of Advanced Manufacturing Technology*, 42(5-6):581, 2009.

[CGRW01] Steven P. Coy, Bruce L. Golden, George C. Runger, and Edward A. Wasil. Using experimental design to find effective parameter settings for heuristics. *J. Heuristics*, 7(1):77–97, 2001.

[CKT91] Peter C. Cheeseman, Bob Kanefsky, and William M. Taylor. Where the really hard problems are. In John Mylopoulos and Raymond Reiter, editors, *Proceedings of the 12th International Joint Conference on Artificial Intelligence. Sydney, Australia, August 24-30, 1991*, pages 331–340. Morgan Kaufmann, 1991.

[CLIHS17] Leslie Pérez Cáceres, Manuel López-Ibáñez, Holger Hoos, and Thomas Stützle. An experimental study of adaptive capping in irace. In *International Conference on Learning and Intelligent Optimization*, pages 235–250. Springer, 2017.

[CLN12] Sivadon Chaisiri, Bu-Sung Lee, and Dusit Niyato. Optimization of resource provisioning cost in cloud computing. *IEEE Transactions on Services Computing*, 5(2):164–177, 2012.

[Coo71] Stephen A Cook. The complexity of theorem-proving procedures. In *Proceedings of the third annual ACM symposium on Theory of computing*, pages 151–158. ACM, 1971.

[CTBP20] Matthias Carnein, Heike Trautmann, Albert Bifet, and Bernhard Pfahringer. confstream: Automated algorithm selection and configuration of stream clustering algorithms. In Ilias S. Kotsireas and Panos M. Pardalos, editors, *Learning and Intelligent Optimization - 14th International Conference, LION 14, Athens, Greece, May 24-28, 2020, Revised Selected Papers*, volume 12096 of *Lecture Notes in Computer Science*, pages 80–95. Springer, 2020.

[CV95] Corinna Cortes and Vladimir Vapnik. Support-vector networks. *Mach. Learn.*, 20(3):273–297, 1995.

[DAD18a] Robinson Duque, Alejandro Arbelaez, and Juan Francisco Díaz. Online over time processing of combinatorial problems. *Constraints An Int. J.*, 23(3):310–334, 2018.

[DAD18b] Robinson Duque, Alejandro Arbelaez, and Juan Francisco Díaz. Processing online SAT instances with waiting time constraints and completion weights. In Giuseppe Nicosia, Panos M. Pardalos, Giovanni Giuffrida, Renato Umeton, and Vincenzo Sciacca, editors, *Machine Learning, Optimization, and Data Science - 4th International Conference, LOD 2018, Volterra, Italy, September 13-16, 2018, Revised Selected Papers*, volume 11331 of *Lecture Notes in Computer Science*, pages 418–430. Springer, 2018.

[Dan65] George Dantzig. *Linear programming and extensions*. Princeton university press, 1965.

[dAP12] Luiz Jonatã Pires de Araújo and Plácido Rogério Pinheiro. A hybrid methodology approach for container loading problem using genetic algorithm to maximize the weight distribution of cargo. In Olympia Roeva, editor, *Real-World Applications of Genetic Algorithms*, chapter 9. IntechOpen, Rijeka, 2012.

[Dar59] Charles Darwin. *On the Origin of Species by Means of Natural Selection*. Murray, London, 1859. or the Preservation of Favored Races in the Struggle for Life.

[DC03] Rina Dechter and David Cohen. *Constraint processing*. Morgan Kaufmann, 2003.

[DCBK18] Hans Degroote, Patrick De Causmaecker, Bernd Bischl, and Lars Kotthoff. A regression-based methodology for online algorithm selection. In Vadim Bulitko and Sabine Storandt, editors, *Proceedings of the Eleventh International Symposium on Combinatorial Search, SOCS 2018, Stockholm, Sweden - 14-15 July 2018*, pages 37–45. AAAI Press, 2018.

[DCMO16] Milan De Cauwer, Deepak Mehta, and Barry O'Sullivan. The temporal bin packing problem: an application to workload management in data centres. In *Tools with Artificial Intelligence (ICTAI), 2016 IEEE 28th International Conference on*, pages 157–164. IEEE, 2016.

[Dev18] Google Developers. The n-queens problem, 2018. [Online; accessed 22-August-2018].

[DLL62] Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem-proving. *Communications of the ACM*, 5(7):394–397, 1962.

[DM94] Rina Dechter and Itay Meiri. Experimental evaluation of preprocessing algorithms for constraint satisfaction problems. *Artificial Intelligence*, 68(2):211–241, 1994.

[DMB08] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

[DMM09] David L Donoho, Arian Maleki, and Andrea Montanari. Message-passing algorithms for compressed sensing. *Proceedings of the National Academy of Sciences*, 106(45):18914–18919, 2009.

[DP60] Martin Davis and Hilary Putnam. A computing procedure for quantification theory. *Journal of the ACM (JACM)*, 7(3):201–215, 1960.

[DVV03] Sven De Vries and Rakesh V Vohra. Combinatorial auctions: A survey. *INFORMS Journal on computing*, 15(3):284–309, 2003.

[EHHL15] Katharina Eggensperger, Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown. Efficient benchmarking of hyperparameter optimizers via surrogates. In Blai Bonet and Sven Koenig, editors, *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence, January 25-30, 2015, Austin, Texas, USA*, pages 1114–1120. AAAI Press, 2015.

[EJKS04] Andreas T Ernst, Houyuan Jiang, Mohan Krishnamoorthy, and David Sier. Staff scheduling and rostering: A review of applications, methods and models. *European journal of operational research*, 153(1):3–27, 2004.

[ELH+18] Katharina Eggensperger, Marius Lindauer, Holger H. Hoos, Frank Hutter, and Kevin Leyton-Brown. Efficient benchmarking of algorithm configurators via model-based surrogates. *Mach. Learn.*, 107(1):15–41, 2018.

[Elo78] Arpad E Elo. *The rating of chessplayers, past and present*, volume 3. Batsford London, 1978.

[ES03] Niklas Eén and Niklas Sörensson. An extensible sat-solver. In *International conference on theory and applications of satisfiability testing*, pages 502–518. Springer, 2003.

[ES11] A. E. Eiben and Selmar K. Smit. Parameter tuning for configuring and

analyzing evolutionary algorithms. *Swarm Evol. Comput.*, 1(1):19–31, 2011.

[ESVH15] Caroline Even, Andreas Schutt, and Pascal Van Hentenryck. A constraint programming approach for non-preemptive evacuation scheduling. In *International Conference on Principles and Practice of Constraint Programming*, pages 574–591. Springer, 2015.

[FD+95] Daniel Frost, Rina Dechter, et al. Look-ahead value ordering for constraint satisfaction problems. In *IJCAI (1)*, pages 572–578. Citeseer, 1995.

[FEF+20] Matthias Feurer, Katharina Eggensperger, Stefan Falkner, Marius Lindauer, and Frank Hutter. Auto-sklearn 2.0: The next generation. *CoRR*, abs/2007.04074, 2020.

[FH16] Chris Fawcett and Holger H. Hoos. Analysing differences between algorithm configurations through ablation. *J. Heuristics*, 22(4):431–458, 2016.

[FKE+15] Matthias Feurer, Aaron Klein, Katharina Eggensperger, Jost Springenberg, Manuel Blum, and Frank Hutter. Efficient and robust automated machine learning. In *Advances in Neural Information Processing Systems*, pages 2962–2970, 2015.

[FLH15] Stefan Falkner, Marius Lindauer, and Frank Hutter. Spysmac: Automated configuration and performance analysis of SAT solvers. In Marijn Heule and Sean A. Weaver, editors, *Theory and Applications of Satisfiability Testing - SAT 2015 - 18th International Conference, Austin, TX, USA, September 24-27, 2015, Proceedings*, volume 9340 of *Lecture Notes in Computer Science*, pages 215–222. Springer, 2015.

[FMO15] Tadhg Fitzgerald, Yuri Malitsky, and Barry O'Sullivan. ReACTR: Real-time Algorithm Configuration through Tournament Rankings. In Qiang Yang and Michael J. Wooldridge, editors, *Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence, IJCAI 2015, Buenos Aires, Argentina, July 25-31, 2015*, pages 304–310. AAAI Press, 2015.

[FMOT14] Tadhg Fitzgerald, Yuri Malitsky, Barry O'Sullivan, and Kevin Tierney. ReACT: Real-Time Algorithm Configuration through Tournaments. In Stefan Edelkamp and Roman Barták, editors, *Proceedings of the Seventh*

Annual Symposium on Combinatorial Search, SOCS 2014, Prague, Czech Republic, 15-17 August 2014. AAAI Press, 2014.

[FO17] Tadhg Fitzgerald and Barry O'Sullivan. Analysing the effect of candidate selection and instance ordering in a realtime algorithm configuration system. In Ahmed Seffah, Birgit Penzenstadler, Carina Alves, and Xin Peng, editors, *Proceedings of the Symposium on Applied Computing, SAC 2017, Marrakech, Morocco, April 3-7, 2017*, pages 1003–1008. ACM, 2017.

[FO19] Tadhg Fitzgerald and Barry O'Sullivan. Candidate selection and instance ordering for realtime algorithm configuration. *Fundam. Informaticae*, 166(2):141–166, 2019.

[FOMT14] Tadhg Fitzgerald, Barry O'Sullivan, Yuri Malitsky, and Kevin Tierney. Online search algorithm configuration. In Carla E. Brodley and Peter Stone, editors, *Proceedings of the Twenty-Eighth AAAI Conference on Artificial Intelligence, July 27 -31, 2014, Québec City, Québec, Canada*, pages 3104–3105. AAAI Press, 2014.

[Fre97] Eugene C Freuder. In pursuit of the holy grail. *Constraints*, 2(1):57–61, 1997.

[FW92] Eugene C Freuder and Richard J Wallace. Partial constraint satisfaction. *Artificial Intelligence*, 58(1-3):21–70, 1992.

[Gas02] William I Gasarch. The p=? np poll. *Sigact News*, 33(2):34–47, 2002.

[Gas12] William I Gasarch. Guest column: The second p=? np poll. *ACM SIGACT News*, 43(2):53–77, 2012.

[GB65] Solomon W Golomb and Leonard D Baumert. Backtrack programming. *Journal of the ACM (JACM)*, 12(4):516–524, 1965.

[Gee92] PA Geelen. Dual viewpoint heuristics for. In *Proc.: ECAI*, volume 92, pages 31–35, 1992.

[GJ99] Mark E Glickman and Albyn C Jones. Rating the chess rating system. *CHANCE-BERLIN THEN NEW YORK-*, 12:21–28, 1999.

[GKNS07] Martin Gebser, Benjamin Kaufmann, André Neumann, and Torsten Schaub. clasp: A conflict-driven answer set solver. In *International Conference on Logic Programming and Nonmonotonic Reasoning*, pages 260–265. Springer, 2007.

[GKS12a] Martin Gebser, Benjamin Kaufmann, and Torsten Schaub. Conflict-driven answer set solving: From theory to practice. *Artif. Intell.*, 187:52–89, 2012.

[GKS12b] Martin Gebser, Benjamin Kaufmann, and Torsten Schaub. Multi-threaded ASP solving with clasp. *Theory Pract. Log. Program.*, 12(4-5):525–545, 2012.

[Gli95] Mark E Glickman. A comprehensive guide to chess ratings. *American Chess Journal*, 3(1):59–102, 1995.

[Gli98] Mark E Glickman. The glicko system. *Boston University*, 1998.

[Gli99] Mark E Glickman. Parameter estimation in large dynamic paired comparison experiments. *Journal of the Royal Statistical Society: Series C (Applied Statistics)*, 48(3):377–394, 1999.

[Gli01] Mark E Glickman. Dynamic paired comparison models with stochastic variances. *Journal of Applied Statistics*, 28(6):673–689, 2001.

[Gli13] Mark E Glickman. Example of the glicko-2 system. *Boston University*, 2013.

[Glo89] Fred Glover. Tabu search—part i. *ORSA Journal on computing*, 1(3):190–206, 1989.

[Glo90a] Fred Glover. Tabu search: A tutorial. *Interfaces*, 20(4):74–94, 1990.

[Glo90b] Fred Glover. Tabu search—part ii. *ORSA Journal on computing*, 2(1):4–32, 1990.

[GMP+01] Ian P. Gent, Ewan MacIntyre, Patrick Prosser, Barbara M. Smith, and Toby Walsh. Random constraint satisfaction: Flaws and structure. *Constraints An Int. J.*, 6(4):345–372, 2001.

[GO20] LLC Gurobi Optimization. Gurobi optimizer reference manual, 2020.

[Gom60] Ralph Gomory. An algorithm for the mixed integer problem. Technical report, RAND CORP SANTA MONICA CA, 1960.

[Gom63] Ralph E Gomory. An algorithm for integer solutions to linear programs. *Recent advances in mathematical programming*, 64:260–302, 1963.

[GRW08] Bruce L Golden, Subramanian Raghavan, and Edward A Wasil. *The vehicle routing problem: latest advances and new challenges*, volume 43. Springer Science & Business Media, 2008.

[GS06] Matteo Gagliolo and Jürgen Schmidhuber. Learning dynamic algorithm portfolios. *Ann. Math. Artif. Intell.*, 47(3-4):295–328, 2006.

[GS11] Matteo Gagliolo and Jürgen Schmidhuber. Algorithm portfolio selection as a bandit problem with unbounded losses. *Ann. Math. Artif. Intell.*, 61(2):49–86, 2011.

[GS17] Adrian Goldwaser and Andreas Schutt. Optimal torpedo scheduling. In *International Conference on Principles and Practice of Constraint Programming*, pages 338–353. Springer, 2017.

[GSS17] François Gonard, Marc Schoenauer, and Michèle Sebag. ASAP.V2 and ASAP.V3: sequential optimization of an algorithm selector and a scheduler. In *Proceedings of the Open Algorithm Selection Challenge 2017, Brussels, Belgium, September 11-12, 2017*, volume 79 of *Proceedings of Machine Learning Research*, pages 8–11. PMLR, 2017.

[GW96] Ian P. Gent and Toby Walsh. Phase transitions and annealed theories: Number partitioning as a case study. In Wolfgang Wahlster, editor, *12th European Conference on Artificial Intelligence, Budapest, Hungary, August 11-16, 1996, Proceedings*, pages 170–174. John Wiley and Sons, Chichester, 1996.

[H+04] David R Hunter et al. Mm algorithms for generalized bradley-terry models. *The annals of statistics*, 32(1):384–406, 2004.

[HE80] Robert M Haralick and Gordon L Elliott. Increasing tree search efficiency for constraint satisfaction problems. *Artificial intelligence*, 14(3):263–313, 1980.

[HE03] Greg Hamerly and Charles Elkan. Learning the k in k-means. In Sebastian Thrun, Lawrence K. Saul, and Bernhard Schölkopf, editors, *Advances in Neural Information Processing Systems 16 [Neural Information Processing Systems, NIPS 2003, December 8-13, 2003, Vancouver and Whistler, British Columbia, Canada]*, pages 281–288. MIT Press, 2003.

[Heb08] Emmanuel Hebrard. Mistral, a constraint satisfaction library. *Proceedings of the Third International CSP Solver Competition*, 3:3, 2008.

[HHL11] Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown. Sequential model-based optimization for general algorithm configuration. In Carlos A. Coello Coello, editor, *Learning and Intelligent Optimization - 5th*

International Conference, LION 5, Rome, Italy, January 17-21, 2011. Selected Papers, volume 6683 of *Lecture Notes in Computer Science*, pages 507–523. Springer, 2011.

[HHL12] Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown. Parallel algorithm configuration. In Youssef Hamadi and Marc Schoenauer, editors, *Learning and Intelligent Optimization - 6th International Conference, LION 6, Paris, France, January 16-20, 2012, Revised Selected Papers*, volume 7219 of *Lecture Notes in Computer Science*, pages 55–70. Springer, 2012.

[HHL13] Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown. Identifying key algorithm parameters and instance features using forward selection. In Giuseppe Nicosia and Panos M. Pardalos, editors, *Learning and Intelligent Optimization - 7th International Conference, LION 7, Catania, Italy, January 7-11, 2013, Revised Selected Papers*, volume 7997 of *Lecture Notes in Computer Science*, pages 364–381. Springer, 2013.

[HHL14] Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown. An efficient approach for assessing hyperparameter importance. In *Proceedings of the 31th International Conference on Machine Learning, ICML 2014, Beijing, China, 21-26 June 2014*, volume 32 of *JMLR Workshop and Conference Proceedings*, pages 754–762. JMLR.org, 2014.

[HHLBS09] Frank Hutter, Holger H Hoos, Kevin Leyton-Brown, and Thomas Stützle. Paramils: an automatic algorithm configuration framework. *Journal of Artificial Intelligence Research*, 36(1):267–306, 2009.

[HHLM09] Frank Hutter, Holger H. Hoos, Kevin Leyton-Brown, and Kevin P. Murphy. An experimental investigation of model-based parameter optimisation: SPO and beyond. In Franz Rothlauf, editor, *Genetic and Evolutionary Computation Conference, GECCO 2009, Proceedings, Montreal, Québec, Canada, July 8-12, 2009*, pages 271–278. ACM, 2009.

[HHLM10] Frank Hutter, Holger H. Hoos, Kevin Leyton-Brown, and Kevin P. Murphy. Time-bounded sequential parameter optimization. In Christian Blum and Roberto Battiti, editors, *Learning and Intelligent Optimization, 4th International Conference, LION 4, Venice, Italy, January 18-22, 2010. Selected Papers*, volume 6073 of *Lecture Notes in Computer Science*, pages 281–298. Springer, 2010.

[HHS07] Frank Hutter, Holger H Hoos, and Thomas Stützle. Automatic algorithm configuration based on local search. In *AAAI*, volume 7, pages 1152–1157, 2007.

[HIK+18] Jason D. Hartline, Nicole Immorlica, Mohammad Reza Khani, Brendan Lucier, and Rad Niazadeh. Fast core pricing for rich advertising auctions. In Éva Tardos, Edith Elkind, and Rakesh Vohra, editors, *Proceedings of the 2018 ACM Conference on Economics and Computation, Ithaca, NY, USA, June 18-22, 2018*, pages 111–112. ACM, 2018.

[HK93] Steven E. Hampson and Dennis F. Kibler. Large plateaus and plateau search in boolean satisfiability problems: When to give up searching and start again. In David S. Johnson and Michael A. Trick, editors, *Cliques, Coloring, and Satisfiability, Proceedings of a DIMACS Workshop, New Brunswick, New Jersey, USA, October 11-13, 1993*, volume 26 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pages 437–455. DIMACS/AMS, 1993.

[HKMO14a] Barry Hurley, Lars Kotthoff, Yuri Malitsky, and Barry O'Sullivan. Proteus: A Hierarchical Portfolio of Solvers and Transformations. In *Proceedings of CPAIOR*, pages 301–317, 2014.

[HKMO14b] Barry Hurley, Lars Kotthoff, Yuri Malitsky, and Barry O'Sullivan. Proteus: A hierarchical portfolio of solvers and transformations. In Helmut Simonis, editor, *Integration of AI and OR Techniques in Constraint Programming - 11th International Conference, CPAIOR 2014, Cork, Ireland, May 19-23, 2014. Proceedings*, volume 8451 of *Lecture Notes in Computer Science*, pages 301–317. Springer, 2014.

[HKMO14c] Barry Hurley, Lars Kotthoff, Yuri Malitsky, and Barry O'Sullivan. Proteus: A hierarchical portfolio of solvers and transformations. In *International Conference on AI and OR Techniques in Constriant Programming for Combinatorial Optimization Problems*, pages 301–317. Springer, 2014.

[HLB+17] Frank Hutter, Marius Lindauer, Adrian Balint, Sam Bayless, Holger Hoos, and Kevin Leyton-Brown. The configurable sat solver challenge (cssc). *Artificial Intelligence*, 243:1–25, 2017.

[HLF+14] Frank Hutter, Manuel López-Ibáñez, Chris Fawcett, Marius Lindauer, Holger H. Hoos, Kevin Leyton-Brown, and Thomas Stützle. Aclib: A

benchmark library for algorithm configuration. In Panos M. Pardalos, Mauricio G. C. Resende, Chrysafis Vogiatzis, and Jose L. Walteros, editors, *Learning and Intelligent Optimization - 8th International Conference, Lion 8, Gainesville, FL, USA, February 16-21, 2014. Revised Selected Papers*, volume 8426 of *Lecture Notes in Computer Science*, pages 36–40. Springer, 2014.

[HLS14] Holger H. Hoos, Marius Lindauer, and Torsten Schaub. claspfolio 2: Advances in algorithm selection for answer set programming. *Theory Pract. Log. Program.*, 14(4-5):569–585, 2014.

[HMG06] Ralf Herbrich, Tom Minka, and Thore Graepel. Trueskill™: A bayesian skill rating system. In *Advances in Neural Information Processing Systems*, pages 569–576, 2006.

[HO01] Nikolaus Hansen and Andreas Ostermeier. Completely derandomized self-adaptation in evolution strategies. *Evolutionary computation*, 9(2):159–195, 2001.

[HO15] Barry Hurley and Barry O'Sullivan. Statistical regimes and runtime prediction. In *IJCAI*, volume 15, pages 318–324, 2015.

[HOO10] Emmanuel Hebrard, Eoin O'Mahony, and Barry O'Sullivan. Constraint programming and combinatorial optimisation in numberjack. In *International Conference on Integration of Artificial Intelligence (AI) and Operations Research (OR) Techniques in Constraint Programming*, pages 181–185. Springer, 2010.

[Hoo12a] H. Hoos. Programming by optimization. *Communications of the ACM*, 55(2):70–80, 2012.

[Hoo12b] Holger H. Hoos. Automated algorithm configuration and parameter tuning. In Youssef Hamadi, Eric Monfroy, and Frédéric Saubion, editors, *Autonomous Search*, pages 37–71. Springer, 2012.

[HT06] Holger H. Hoos and Edward P. K. Tsang. Local search methods. In Francesca Rossi, Peter van Beek, and Toby Walsh, editors, *Handbook of Constraint Programming*, volume 2 of *Foundations of Artificial Intelligence*, pages 135–167. Elsevier, 2006.

[Hur16] Barry Hurley. *Exploiting machine learning for combinatorial problem solving and optimisation*. PhD thesis, 2016.

[HXHLB14] Frank Hutter, Lin Xu, Holger H Hoos, and Kevin Leyton-Brown. Algorithm runtime prediction: Methods & evaluation. *Artificial Intelligence*, 206:79–111, 2014.

[IBM14] IBM, 2014. IBM ILOG CPLEX Optimization Studio 12.6.1.

[JLBRS12] Matti Järvisalo, Daniel Le Berre, Olivier Roussel, and Laurent Simon. The international sat solver competitions. *AI Magazine*, 33(1):89–92, 2012.

[JMZ+15] Lu Jiang, Deyu Meng, Qian Zhao, Shiguang Shan, and Alexander G. Hauptmann. Self-paced curriculum learning. In Blai Bonet and Sven Koenig, editors, *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence, January 25-30, 2015, Austin, Texas, USA*, pages 2694–2700. AAAI Press, 2015.

[KCL17] Anders Nicolai Knudsen, Marco Chiarandini, and Kim S Larsen. Constraint handling in flight planning. In *International Conference on Principles and Practice of Constraint Programming*, pages 354–369. Springer, 2017.

[KGK16] Yubo Kou, Xinning Gui, and Yong Ming Kow. Ranking practices and distinction in league of legends. In *Proceedings of the 2016 Annual Symposium on Computer-Human Interaction in Play*, pages 4–9, 2016.

[KGV83] Scott Kirkpatrick, C Daniel Gelatt, and Mario P Vecchi. Optimization by simulated annealing. *science*, 220(4598):671–680, 1983.

[KHNT19] Pascal Kerschke, Holger H. Hoos, Frank Neumann, and Heike Trautmann. Automated algorithm selection: Survey and perspectives. *Evol. Comput.*, 27(1):3–45, 2019.

[KHR+02] Henry A. Kautz, Eric Horvitz, Yongshao Ruan, Carla P. Gomes, and Bart Selman. Dynamic restart policies. In Rina Dechter, Michael J. Kearns, and Richard S. Sutton, editors, *Proceedings of the Eighteenth National Conference on Artificial Intelligence and Fourteenth Conference on Innovative Applications of Artificial Intelligence, July 28 - August 1, 2002, Edmonton, Alberta, Canada*, pages 674–681. AAAI Press / The MIT Press, 2002.

[KKHT15] Lars Kotthoff, Pascal Kerschke, Holger H. Hoos, and Heike Trautmann. Improving the state of the art in inexact TSP solving using per-instance algorithm selection. In Clarisse Dhaenens, Laetitia Jourdan, and Marie-

Eléonore Marmion, editors, *Learning and Intelligent Optimization - 9th International Conference, LION 9, Lille, France, January 12-15, 2015. Revised Selected Papers*, volume 8994 of *Lecture Notes in Computer Science*, pages 202–217. Springer, 2015.

[KLBL17] Robert Kleinberg, Kevin Leyton-Brown, and Brendan Lucier. Efficiency through procrastination: Approximately optimal algorithm configuration with runtime guarantees. In *Proceedings of the International Joint Conference on Artificial Intelligence*, 2017.

[KLBLG19] Robert Kleinberg, Kevin Leyton-Brown, Brendan Lucier, and Devon Graham. Procrastinating with confidence: Near-optimal, anytime, adaptive algorithm configuration. *arXiv preprint arXiv:1902.05454*, 2019.

[KMS+11] Serdar Kadioglu, Yuri Malitsky, Ashish Sabharwal, Horst Samulowitz, and Meinolf Sellmann. Algorithm selection and scheduling. In Jimmy Ho-Man Lee, editor, *Principles and Practice of Constraint Programming - CP 2011 - 17th International Conference, CP 2011, Perugia, Italy, September 12-16, 2011. Proceedings*, volume 6876 of *Lecture Notes in Computer Science*, pages 454–469. Springer, 2011.

[KMST10] S. Kadioglu, Y. Malitsky, M. Sellmann, and K. Tierney. ISAC – Instance-Specific Algorithm Configuration. *ECAI*, pages 751–756, 2010.

[Kot14] Lars Kotthoff. Algorithm selection for combinatorial search problems: A survey. *AI Mag.*, 35(3):48–60, 2014.

[KP14] Volodymyr Kuleshov and Doina Precup. Algorithms for multi-armed bandit problems. *arXiv preprint arXiv:1402.6028*, 2014.

[KPK10] M. Pawan Kumar, Benjamin Packer, and Daphne Koller. Self-paced learning for latent variable models. In John D. Lafferty, Christopher K. I. Williams, John Shawe-Taylor, Richard S. Zemel, and Aron Culotta, editors, *Advances in Neural Information Processing Systems 23: 24th Annual Conference on Neural Information Processing Systems 2010. Proceedings of a meeting held 6-9 December 2010, Vancouver, British Columbia, Canada*, pages 1189–1197. Curran Associates, Inc., 2010.

[KR+10] Miron B Kursa, Witold R Rudnicki, et al. Feature selection with the boruta package. *J Stat Softw*, 36(11):1–13, 2010.

[KTH+17] Lars Kotthoff, Chris Thornton, Holger H. Hoos, Frank Hutter, and Kevin Leyton-Brown. Auto-weka 2.0: Automatic model selection and hy-

perparameter optimization in WEKA. *Journal of Machine Learning Research*, 18:25:1–25:5, 2017.

[Kum13] Anit Kumar. Encoding schemes in genetic algorithm. *International Journal of Advanced Research in IT and Engineering*, 2(3):1–7, 2013.

[Kur17] Miron Bartosz Kursa. Efficient all relevant feature selection with random ferns. In Marzena Kryszkiewicz, Annalisa Appice, Dominik Slezak, Henryk Rybinski, Andrzej Skowron, and Zbigniew W. Ras, editors, *Foundations of Intelligent Systems - 23rd International Symposium, ISMIS 2017, Warsaw, Poland, June 26-29, 2017, Proceedings*, volume 10352 of *Lecture Notes in Computer Science*, pages 302–311. Springer, 2017.

[LAMG20] Tong Liu, Roberto Amadini, Jacopo Mauro, and Maurizio Gabbrielli. sunny-as2: Enhancing SUNNY for algorithm selection. *CoRR*, abs/2009.03107, 2020.

[LBMS17] Kevin Leyton-Brown, Paul Milgrom, and Ilya Segal. Economics and computer science of a radio spectrum reallocation. *Proceedings of the National Academy of Sciences*, 114(28):7202–7209, 2017.

[LBPS00a] K. Leyton-Brown, M. Pearson, and Y. Shoham. Towards a universal test suite for combinatorial auction algorithms. In *Proceedings of the 2nd ACM conference on Electronic commerce*, pages 66–76. ACM, 2000.

[LBPS00b] Kevin Leyton-Brown, Mark Pearson, and Yoav Shoham. Towards a universal test suite for combinatorial auction algorithms. In *Proceedings of the 2nd ACM conference on Electronic commerce*, pages 66–76. ACM, 2000.

[LBPS00c] Kevin Leyton-Brown, Mark Pearson, and Yoav Shoham. Towards a universal test suite for combinatorial auction algorithms. In *Proceedings ACM-EC*, pages 66–76. ACM, 2000.

[LBS03] Daniel Le Berre and Laurent Simon. The essentials of the sat 2003 competition. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 452–467. Springer, 2003.

[LBS04] Daniel Le Berre and Laurent Simon. Fifty-five solvers in vancouver: The sat 2004 competition. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 321–344. Springer, 2004.

[LCH+14] Canhong Lin, King Lun Choy, George TS Ho, Sai Ho Chung, and HY Lam. Survey of green vehicle routing problem: past and future trends. *Expert Systems with Applications*, 41(4):1118–1138, 2014.

[LFC03] Eva K Lee, Tim Fox, and Ian Crocker. Integer programming applied to intensity-modulated radiation therapy treatment planning. *Annals of Operations Research*, 119(1-4):165–181, 2003.

[LH18] Marius Lindauer and Frank Hutter. Warmstarting of model-based algorithm configuration. In Sheila A. McIlraith and Kilian Q. Weinberger, editors, *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence, (AAAI-18), the 30th innovative Applications of Artificial Intelligence (IAAI-18), and the 8th AAAI Symposium on Educational Advances in Artificial Intelligence (EAAI-18), New Orleans, Louisiana, USA, February 2-7, 2018*, pages 1355–1362. AAAI Press, 2018.

[LHHS17] Marius Lindauer, Frank Hutter, Holger H. Hoos, and Torsten Schaub. Autofolio: An automatically configured algorithm selector (extended abstract). In Carles Sierra, editor, *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI 2017, Melbourne, Australia, August 19-25, 2017*, pages 5025–5029. ijcai.org, 2017.

[LIDLC+16] Manuel López-Ibáñez, Jérémie Dubois-Lacoste, Leslie Pérez Cáceres, Mauro Birattari, and Thomas Stützle. The irace package: Iterated racing for automatic algorithm configuration. *Operations Research Perspectives*, 3:43–58, 2016.

[LNS09] Kevin Leyton-Brown, Eugene Nudelman, and Yoav Shoham. Empirical hardness models: Methodology and a case study on combinatorial auctions. *J. ACM*, 56(4):22:1–22:52, 2009.

[LRT14] Balaji Lakshminarayanan, Daniel M. Roy, and Yee Whye Teh. Mondrian forests: Efficient online random forests. In Zoubin Ghahramani, Max Welling, Corinna Cortes, Neil D. Lawrence, and Kilian Q. Weinberger, editors, *Advances in Neural Information Processing Systems 27: Annual Conference on Neural Information Processing Systems 2014, December 8-13 2014, Montreal, Quebec, Canada*, pages 3140–3148, 2014.

[LvRK19] Marius Lindauer, Jan N. van Rijn, and Lars Kotthoff. The algorithm selection competitions 2015 and 2017. *Artif. Intell.*, 272:86–100, 2019.

[M+75] Gordon E Moore et al. Progress in digital integrated electronics. In *Electron Devices Meeting*, volume 21, pages 11–13, 1975.

[Mac77] Alan K. Mackworth. Consistency in networks of relations. *Artif. Intell.*, 8(1):99–118, 1977.

[MB13] Frank Mugrauer and Adrian Balint. Sat encoded graph isomorphism (gi) benchmark description. *Proceedings of SAT Competition 2013*, page 115, 2013.

[MJPL92] Steven Minton, Mark D Johnston, Andrew B Philips, and Philip Laird. Minimizing conflicts: a heuristic repair method for constraint satisfaction and scheduling problems. *Artificial Intelligence*, 58(1-3):161–205, 1992.

[MJSS16] David R. Morrison, Sheldon H. Jacobson, Jason J. Sauppe, and Edward C. Sewell. Branch-and-bound algorithms: A survey of recent advances in searching, branching, and pruning. *Discret. Optim.*, 19:79–102, 2016.

[ML16] Norbert Manthey and Marius Lindauer. Spybug: Automated bug detection in the configuration space of SAT solvers. In Nadia Creignou and Daniel Le Berre, editors, *Theory and Applications of Satisfiability Testing - SAT 2016 - 19th International Conference, Bordeaux, France, July 5-8, 2016, Proceedings*, volume 9710 of *Lecture Notes in Computer Science*, pages 554–561. Springer, 2016.

[MM88] Roger Mohr and Gérald Masini. Good old discrete relaxation. In *Proceedings of the 8th European conference on artificial intelligence*, pages 651–656, 1988.

[MMO13] Yuri Malitsky, Deepak Mehta, and Barry O'Sullivan. Evolving instance specific algorithm configuration. In *Sixth Annual Symposium on Combinatorial Search*, 2013.

[MMZ+01] Matthew W Moskewicz, Conor F Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient sat solver. In *Proceedings of the 38th annual Design Automation Conference*, pages 530–535. ACM, 2001.

[Mor93] Paul Morris. The breakout method for escaping from local minima. In *AAAI*, volume 93, pages 40–45, 1993.

[MS14] Norbert Manthey and Peter Steinke. Too many rooks. *Proceedings of SAT Competition 2014*, page 97, 2014.

[MSSS13a] Yuri Malitsky, Ashish Sabharwal, Horst Samulowitz, and Meinolf Sellmann. Algorithm portfolios based on cost-sensitive hierarchical clustering. In Francesca Rossi, editor, *IJCAI 2013, Proceedings of the 23rd International Joint Conference on Artificial Intelligence, Beijing, China, August 3-9, 2013*, pages 608–614. IJCAI/AAAI, 2013.

[MSSS13b] Yuri Malitsky, Ashish Sabharwal, Horst Samulowitz, and Meinolf Sellmann. Algorithm portfolios based on cost-sensitive hierarchical clustering. In Francesca Rossi, editor, *IJCAI 2013, Proceedings of the 23rd International Joint Conference on Artificial Intelligence, Beijing, China, August 3-9, 2013*, pages 608–614. IJCAI/AAAI, 2013.

[MWB+20] Adil El Mesaoudi-Paul, Dimitri Weiß, Viktor Bengs, Eyke Hüllermeier, and Kevin Tierney. Pool-based realtime algorithm configuration: A preselection bandit approach. In Ilias S. Kotsireas and Panos M. Pardalos, editors, *Learning and Intelligent Optimization - 14th International Conference, LION 14, Athens, Greece, May 24-28, 2020, Revised Selected Papers*, volume 12096 of *Lecture Notes in Computer Science*, pages 216–232. Springer, 2020.

[Nad90] Bernard A. Nadel. Representation selection for constraint satisfaction: A case study using n-queens. *IEEE Expert*, 5(3):16–23, 1990.

[Nis00] Noam Nisan. Bidding and allocation in combinatorial auctions. In *Proceedings of the 2nd ACM conference on Electronic commerce*, pages 1–12. ACM, 2000.

[NPBT07] Roland Nilsson, José M. Peña, Johan Björkegren, and Jesper Tegnér. Consistent feature selection for pattern recognition in polynomial time. *J. Mach. Learn. Res.*, 8:589–612, 2007.

[NSB+07a] Nicholas Nethercote, Peter J Stuckey, Ralph Becket, Sebastian Brand, Gregory J Duck, and Guido Tack. Minizinc: Towards a standard cp modelling language. In *International Conference on Principles and Practice of Constraint Programming*, pages 529–543. Springer, 2007.

[NSB+07b] Nicholas Nethercote, Peter J. Stuckey, Ralph Becket, Sebastian Brand, Gregory J. Duck, and Guido Tack. Minizinc: Towards a standard CP modelling language. In Christian Bessiere, editor, *Principles and Practice of Constraint Programming - CP 2007, 13th International Conference, CP 2007, Providence, RI, USA, September 23-27, 2007, Proceedings*,

volume 4741 of *Lecture Notes in Computer Science*, pages 529–543. Springer, 2007.

[OHH+08a] Eoin O'Mahony, Emmanuel Hebrard, Alan Holland, Conor Nugent, and Barry O'Sullivan. Using case-based reasoning in an algorithm portfolio for constraint solving. In *Irish conference on artificial intelligence and cognitive science*, pages 210–216, 2008.

[OHH+08b] Eoin O'Mahony, Emmanuel Hebrard, Alan Holland, Conor Nugent, and Barry O'Sullivan. Using case-based reasoning in an algorithm portfolio for constraint solving. In *Irish conference on artificial intelligence and cognitive science*, pages 210–216, 2008.

[PBG05] Mukul R. Prasad, Armin Biere, and Aarti Gupta. A survey of recent advances in sat-based formal verification. *Int. J. Softw. Tools Technol. Transf.*, 7(2):156–173, 2005.

[PFCO15] Steven D Prestwich, Adejuyigbe O Fajemisin, Laura Climent, and Barry O'Sullivan. Solving a hard cutting stock problem by machine learning and optimisation. In *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*, pages 335–347. Springer, 2015.

[PFL17] Charles Prud'homme, Jean-Guillaume Fages, and Xavier Lorca. *Choco Documentation*. TASC - LS2N CNRS UMR 6241, COSLING S.A.S., 2017.

[PH18] Yasha Pushak and Holger H. Hoos. Algorithm configuration landscapes: - more benign than expected? In Anne Auger, Carlos M. Fonseca, Nuno Lourenço, Penousal Machado, Luís Paquete, and L. Darrell Whitley, editors, *Parallel Problem Solving from Nature - PPSN XV - 15th International Conference, Coimbra, Portugal, September 8-12, 2018, Proceedings, Part II*, volume 11102 of *Lecture Notes in Computer Science*, pages 271–283. Springer, 2018.

[PH20] Yasha Pushak and Holger H. Hoos. Golden parameter search: exploiting structure to quickly configure parameters in parallel. In Carlos Artemio Coello Coello, editor, *GECCO '20: Genetic and Evolutionary Computation Conference, Cancún Mexico, July 8-12, 2020*, pages 245–253. ACM, 2020.

[PJS20] Guido Tack Peter J. Stuckey, Kim Marriott. Minizinc tutorial, 2020.

[PPdSN19] Rafael Stubs Parpinelli, Guilherme Plichoski, Renan Samuel da Silva,

and Pedro Henrique Narloch. A review of techniques for online control of parameters in swarm intelligence and evolutionary computation algorithms. *Int. J. Bio Inspired Comput.*, 13(1):1–20, 2019.

[PRL+14] Marie Pelleau, Louis-Martin Rousseau, Pierre L'Ecuyer, Walid Zegal, and Louis Delorme. Scheduling agents using forecast call arrivals at hydro-quebec's call centers. In *International Conference on Principles and Practice of Constraint Programming*, pages 862–869. Springer, 2014.

[PVG+11] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[PW06] Yves Pochet and Laurence A Wolsey. *Production planning by mixed integer programming*. Springer Science & Business Media, 2006.

[Ran71] William M Rand. Objective criteria for the evaluation of clustering methods. *Journal of the American Statistical association*, 66(336):846–850, 1971.

[Res09] Mauricio G. C. Resende. Greedy randomized adaptive search procedures. In Christodoulos A. Floudas and Panos M. Pardalos, editors, *Encyclopedia of Optimization, Second Edition*, pages 1460–1469. Springer, 2009.

[RF12] Olympia Roeva and Stefka Fidanova. Application of genetic algorithms and ant colony optimization for modelling of e. coli cultivation process. In Olympia Roeva, editor, *Real-World Applications of Genetic Algorithms*, chapter 13. IntechOpen, Rijeka, 2012.

[Ric76] John R. Rice. The algorithm selection problem. *Adv. Comput.*, 15:65–118, 1976.

[RJ07] Filip Radlinski and Thorsten Joachims. Active exploration for learning rankings from clickthrough data. In *Proceedings of the 13th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 570–579. ACM, 2007.

[RN16] Stuart J Russell and Peter Norvig. *Artificial intelligence: a modern approach*. Malaysia; Pearson Education Limited,, 2016.

[Rup18] Karl Rupp. *42 Years of Microprocessor Trend Data*. Feb 2018.

[SBA+20] Gresa Shala, André Biedenkapp, Noor Awad, Steven Adriaensen, Marius Lindauer, and Frank Hutter. Learning step-size adaptation in CMA-ES. In Thomas Bäck, Mike Preuss, André H. Deutz, Hao Wang, Carola Doerr, Michael T. M. Emmerich, and Heike Trautmann, editors, *Parallel Problem Solving from Nature - PPSN XVI - 16th International Conference, PPSN 2020, Leiden, The Netherlands, September 5-9, 2020, Proceedings, Part I*, volume 12269 of *Lecture Notes in Computer Science*, pages 691–706. Springer, 2020.

[SBH+20] David Speck, André Biedenkapp, Frank Hutter, Robert Mattmüller, and Marius Lindauer. Learning heuristic selection with dynamic algorithm configuration. *CoRR*, abs/2006.08246, 2020.

[Sch99] T Schoning. A probabilistic algorithm for k-sat and constraint satisfaction problems. In *Foundations of Computer Science, 1999. 40th Annual Symposium on*, pages 410–414. IEEE, 1999.

[SH13] James Styles and Holger H. Hoos. Ordered racing protocols for automatically configuring algorithms for scaling performance. In Christian Blum and Enrique Alba, editors, *Genetic and Evolutionary Computation Conference, GECCO '13, Amsterdam, The Netherlands, July 6-10, 2013*, pages 551–558. ACM, 2013.

[SHM12] James Styles, Holger H. Hoos, and Martin Müller. Automatically configuring algorithms for scaling performance. In Youssef Hamadi and Marc Schoenauer, editors, *Learning and Intelligent Optimization - 6th International Conference, LION 6, Paris, France, January 16-20, 2012, Revised Selected Papers*, volume 7219 of *Lecture Notes in Computer Science*, pages 205–219. Springer, 2012.

[Sim05] Helmut Simonis. Sudoku as a constraint problem. In *CP Workshop on modeling and reformulating Constraint Satisfaction Problems*, volume 12, pages 13–27. Citeseer, 2005.

[SK93] Bart Selman and Henry Kautz. Domain-independent extensions to gsat: Solving large structured satisfiability problems. In *IJCAI*, volume 93, pages 290–295. Citeseer, 1993.

[SKC94] Bart Selman, Henry A Kautz, and Bram Cohen. Noise strategies for improving local search. In *AAAI*, volume 94, pages 337–343, 1994.

[SLBH05] Laurent Simon, Daniel Le Berre, and Edward A Hirsch. The sat2002 competition. volume 43, pages 307–342. Springer, 2005.

[SLM+92] Bart Selman, Hector J Levesque, David G Mitchell, et al. A new method for solving hard satisfiability problems. In *AAAI*, volume 92, pages 440–446. Citeseer, 1992.

[Smi08] Kate Smith-Miles. Cross-disciplinary perspectives on meta-learning for algorithm selection. *ACM Comput. Surv.*, 41(1):6:1–6:25, 2008.

[SP94] M. Srinivas and Lalit M. Patnaik. Genetic algorithms: A survey. *Computer*, 27(6):17–26, 1994.

[SS87] Harold S Stone and Janice M Stone. Efficient search techniques—an empirical study of the n-queens problem. *IBM Journal of Research and Development*, 31(4):464–474, 1987.

[SS96] J. P. Marques Silva and K. A. Sakallah. Grasp-a new search algorithm for satisfiability. In *Proceedings of International Conference on Computer Aided Design*, pages 220–227, 1996.

[SSHH15] Jendrik Seipp, Silvan Sievers, Malte Helmert, and Frank Hutter. Automatic configuration of sequential planning portfolios. In Blai Bonet and Sven Koenig, editors, *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence, January 25-30, 2015, Austin, Texas, USA*, pages 3364–3370. AAAI Press, 2015.

[TARC16] Charlotte Truchet, Alejandro Arbelaez, Florian Richoux, and Philippe Codognet. Estimating parallel runtimes for randomized algorithms in constraint solving. *J. Heuristics*, 22(4):613–648, 2016.

[TBH11] Dave A. D. Tompkins, Adrian Balint, and Holger H. Hoos. Captain jack: New variable selection heuristics in local search for SAT. In Karem A. Sakallah and Laurent Simon, editors, *Theory and Applications of Satisfiability Testing - SAT 2011 - 14th International Conference, SAT 2011, Ann Arbor, MI, USA, June 19-22, 2011. Proceedings*, volume 6695 of *Lecture Notes in Computer Science*, pages 302–316. Springer, 2011.

[THHL13] Chris Thornton, Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown. Auto-weka: combined selection and hyperparameter optimization of classification algorithms. In Inderjit S. Dhillon, Yehuda Koren, Rayid Ghani, Ted E. Senator, Paul Bradley, Rajesh Parekh, Jingrui He, Robert L. Grossman, and Ramasamy Uthurusamy, editors, *The 19th ACM SIGKDD*

International Conference on Knowledge Discovery and Data Mining, KDD 2013, Chicago, IL, USA, August 11-14, 2013, pages 847–855. ACM, 2013.

[TLZ+05] Li Tang, Jun Li, Jin Zhou, Zhizhi Zhou, Hao Wang, and Kai Li. Freerank: implementing independent ranking service for multiplayer online games. In *Proceedings of 4th ACM SIGCOMM workshop on Network and system support for games*, pages 1–7, 2005.

[Tor13] Jacobo Torán. On the resolution complexity of graph non-isomorphism. In Matti Järvisalo and Allen Van Gelder, editors, *Theory and Applications of Satisfiability Testing - SAT 2013 - 16th International Conference, Helsinki, Finland, July 8-12, 2013. Proceedings*, volume 7962 of *Lecture Notes in Computer Science*, pages 52–66. Springer, 2013.

[TV14] Paolo Toth and Daniele Vigo. *Vehicle routing: problems, methods, and applications*. SIAM, 2014.

[vB06] Peter van Beek. Backtracking search algorithms. In Francesca Rossi, Peter van Beek, and Toby Walsh, editors, *Handbook of Constraint Programming*, volume 2 of *Foundations of Artificial Intelligence*, pages 85–134. Elsevier, 2006.

[VCT13] Philippe Vismara, Remi Coletta, and Gilles Trombettoni. Constrained wine blending. In *International Conference on Principles and Practice of Constraint Programming*, pages 864–879. Springer, 2013.

[VM05] Joannes Vermorel and Mehryar Mohri. Multi-armed bandit algorithms and empirical evaluation. In *Proceedings of ECML*, pages 437–448. Springer, 2005.

[Wed95] Dag Wedelin. An algorithm for large scale 0–1 integer programming with application to airline crew scheduling. *Annals of operations research*, 57(1):283–301, 1995.

[WGS18] Gellért Weisz, András György, and Csaba Szepesvári. Leapsandbounds: A method for approximately optimal algorithm configuration. *arXiv preprint arXiv:1807.00755*, 2018.

[WGS19] Gellért Weisz, András György, and Csaba Szepesvári. Capsandruns: An improved method for approximately optimal algorithm configuration. In Kamalika Chaudhuri and Ruslan Salakhutdinov, editors, *Proceedings of the 36th International Conference on Machine Learning, ICML 2019, 9-*

15 June 2019, Long Beach, California, USA, volume 97 of *Proceedings of Machine Learning Research*, pages 6707–6715. PMLR, 2019.

[Wol98] Laurence A Wolsey. *Integer Programming. Series in Discrete Mathematics and Optimization*. Wiley-Interscience New Jersey, 1998.

[XHHL12] Lin Xu, Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown. Evaluating component solver contributions to portfolio-based algorithm selectors. In Alessandro Cimatti and Roberto Sebastiani, editors, *Theory and Applications of Satisfiability Testing - SAT 2012 - 15th International Conference, Trento, Italy, June 17-20, 2012. Proceedings*, volume 7317 of *Lecture Notes in Computer Science*, pages 228–241. Springer, 2012.

[XHHLB08] Lin Xu, Frank Hutter, Holger H Hoos, and Kevin Leyton-Brown. Satzilla: portfolio-based algorithm selection for sat. *Journal of artificial intelligence research*, 32:565–606, 2008.

[XHHLB09] Lin Xu, Frank Hutter, Holger H Hoos, and Kevin Leyton-Brown. Satzilla2009: an automatic algorithm portfolio for sat. *SAT*, 4:53–55, 2009.

[XHHLB12] Lin Xu, Frank Hutter, Holger Hoos, and Kevin Leyton-Brown. Features for sat. *University of British Columbia,, Tech. Rep*, 2012.

[XHL10] Lin Xu, Holger H. Hoos, and Kevin Leyton-Brown. Hydra: Automatically configuring algorithms for portfolio-based selection. In Maria Fox and David Poole, editors, *Proceedings of the Twenty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2010, Atlanta, Georgia, USA, July 11-15, 2010*. AAAI Press, 2010.

[XHS+12] L. Xu, F. Hutter, J. Shen, H.H. Hoos, and K. Leyton-Brown. Satzilla2012: Improved algorithm selection based on cost-sensitive classification models, 2012. SAT Competition.

[Zer29] Ernst Zermelo. Die berechnung der turnier-ergebnisse als ein maximumproblem der wahrscheinlichkeitsrechnung. *Mathematische Zeitschrift*, 29(1):436–460, 1929.

[Zon14] Doug Zongker. trueskill.py, 2014. https://github.com/dougz/trueskill.